Lecture Notes in Computer Science 2884
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Elie Najm Uwe Nestmann
Perdita Stevens (Eds.)

Formal Methods for Open Object-Based Distributed Systems

6th IFIP WG 6.1 International Conference, FMOODS 2003
Paris, France, November 19-21, 2003
Proceedings

Springer

Series Editors

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

Volume Editors

Elie Najm
ENST, Dépt. Informatique et Réseaux
46, rue Barrault, 75634 Paris, France
E-mail: Elie.Najm@enst.fr

Uwe Nestmann
EPFL-I and C
Ecublens INR 317, 1015 Lausanne, Switzerland
E-mail: uwe.nestmann@EPFL.ch

Perdita Stevens
University of Edinburgh, Laboratory for Foundations of Computer Science
JCMB, King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK
E-mail: perdita.stevens@ed.ac.uk

Cataloging-in-Publication Data applied for

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

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

CR Subject Classification (1998): C.2.4, D.1.3, D.2, D.3, F.3, D.4

ISSN 0302-9743
ISBN 3-540-20491-1 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 10967973 06/3142 5 4 3 2 1 0

Preface

This volume contains the proceedings of FMOODS 2003, the 6 th IFIP WG 6.1 International Conference on *Formal Methods for Open Object-Based Distributed Systems*. The conference was held in Paris, France on November 19–21, 2003. The event was the sixth meeting of this conference series, which is held roughly every year and a half, the earlier events having been held in Paris, Canterbury, Florence, Stanford, and Twente.

The goal of the FMOODS series of conferences is to bring together researchers whose work encompasses three important and related fields:

- formal methods;
- distributed systems;
- object-based technology.

Such a convergence is representative of recent advances in the field of distributed systems, and provides links between several scientific and technological communities, as represented by the conferences FORTE/PSTV, CONCUR, and ECOOP.

The objective of FMOODS is to provide an integrated forum for the presentation of research in the above-mentioned fields, and the exchange of ideas and experiences in the topics concerned with the formal methods support for open object-based distributed systems. For the call for papers, aspects of interest of the considered systems included, but were not limited to: formal models; formal techniques for specification, design or analysis; component-based design; verification, testing and validation; semantics of programming, coordination, or modeling languages; type systems for programming, coordination or modelling languages; behavioral typing; multiple viewpoint modelling and consistency between different models; transformations of models; integration of quality of service requirements into formal models; formal models for security; and applications and experience, carefully described. Work on these aspects of (official and de facto) standard notations and languages, e.g., the UML, and on component-based design, was explicitly welcome.

In total 78 abstracts and 63 papers were submitted to this year's conference, covering the full range of topics listed above. Out of the submissions, 18 research papers were selected by the program committee for presentation. We would like to express our deepest appreciation to the authors of all submitted papers and to the program committee members and external reviewers who did an outstanding job in selecting the best papers for presentation.

For the first time, the FMOODS conference was held as a joint event in federation with the 4th IFIP WG 6.1 International Conference on *Distributed Applications and Interoperable Systems* (DAIS 2003). The co-location of the 2003 vintages of the FMOODS and DAIS conferences provided an excellent opportunity to the participants for a wide and comprehensive exchange of ideas within the domain of distributed systems and applications. Both FMOODS and DAIS address this domain, the former with its emphasis on formal approaches, the latter on practical solutions. Their combination in a single event ensured that both theoretical foundations and practical issues were presented and discussed. Also due to the federation of the two conferences, the topics of reconfigurability and component-based design were particularly emphasized this year, along with the many open issues related to openness and interoperability of distributed systems and applications. To further the interaction between the two communities, participants to the federated event were offered a single registration procedure and were entitled to choose freely between DAIS and FMOODS sessions. Also, several invited speakers were explicitly scheduled as joint sessions. As another novelty, this year's conference included a two-day tutorial and workshop session, the latter again explicitly held as a joint event. Details can be found at the conference website: `http://fedconf.enst.fr/`.

Special thanks to Michel Riguidel, head of the Networks and Computer Science department of ENST. His support made this event happen. We would also like to thank Lynne Blair who chaired the workshop selection process, and Sylvie Vignes who chaired the tutorial selection process. We are grateful to David Chambliss, Andrew Herbert, Bart Jacobs, Bertrand Meyer, and Alan Cameron Wills for agreeing to present invited talks at the conference.

We thank Jennifer Tenzer for help with running the electronic submission and conference management system, and the Laboratory for Foundations of Computer Science at the University of Edinburgh for financially supporting this help. We used CyberChair (http://www.cyberchair.org); we thank Julian Bradfield for advice on adapting it for our particular needs. As of today, we have also received sponsorships from CNRS-ARP, GET, EDF, ENST, and INRIA. Other contributors are also expected. We extend our thanks to all of them.

We thank Laurent Pautet for acting as Local Arrangements Chair and John Derrick for his work as Publicity Chair. We would also like to thank the FMOODS Steering Committee members for their advice.

September 2003

Elie Najm
FMOODS 2003 General Chair

Uwe Nestmann
Perdita Stevens
FMOODS 2003 Program Chairs

Organization

General Chair	Elie Najm (ENST, France)
Program Chairs	Uwe Nestmann (EPFL, Switzerland)
	Perdita Stevens (University of Edinburgh, UK)
Tutorial Chair	Sylvie Vignes (ENST, France)
Workshop Chair	Lynne Blair (Lancaster University, UK)
Local Organization Chair	Laurent Pautet (ENST, France)
Publicity Chair	John Derrick (University of Kent, UK)

Steering Committee

John Derrick
Roberto Gorrieri
Guy Leduc
Elie Najm

Program Committee

Lynne Blair (UK)
Michele Bugliesi (Italy)
Denis Caromel (France)
John Derrick (UK)
Alessandro Fantechi (Italy)
Kokichi Futatsugi (Japan)
Andy Gordon (UK)
Cosimo Laneve (Italy)
Luigi Logrippo (Canada)
Elie Najm (France)
Erik Poll (The Netherlands)
Arend Rensink (The Netherlands)
Bernhard Rumpe (Germany)
Martin Steffen (Germany)
Carolyn Talcott (USA)
Nalini Venkatasubramanian (USA)

Referees

Erika Ábrahám
Dave Akehurst
Isabelle Attali
Arnaud Bailly
Paolo Baldan
Tomás Barros
Benoit Baudry
Clara Benac Earle
Nick Benton
Kirill Bogdanov
Tommaso Bolognesi
Rabea Boulifa
Julian Bradfield
Sebastien Briais
M. C. Bujorianu
Daniel C. Bünzli
Nadia Busi
Arnaud Contes
Steve Cook
Grit Denker
Vijay D'silva
Juan Guillen Scholten
Sebastian Gutierrez-Nolasco
Ludovic Henrio
Jozef Hooman
Fabrice Huet
Ralf Huuck
Bart Jacobs
Ferhat Khendek
Juliana Küster Filipe
Marcel Kyas
Giuseppe Lami
Diego Latella
Giuseppe Lipari
Luigi Liquori
Felipe Luna
Eric Madelaine
Ian Mason
Mieke Massink
Nikolay Mihaylov
Shivajit Mohapatra
Rocco Moretti
Kazuki Munakata
Masaki Nakamura
Masahiro Nakano
Kazuhiro Ogata
Martijn Oostdijk
Carla Piazza
Fabio Pittarello
Alberto Pravato
Antonio Ravara
Bernhard Reus
Ant Rowstron
Takahiro Seino
Don Syme
Alexandre Tauveron
Jennifer Tenzer
Lucian Wischik
Jianwen Xiang
Gianluigi Zavattaro
Matthias Zenger

Table of Contents

Java's Integral Types in PVS

Bart Jacobs

Dep. Comp. Sci., Univ. Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
bart@cs.kun.nl
www.cs.kun.nl/~bart

Abstract. This paper presents an extension of the standard bitvector library of the theorem prover PVS with multiplication, division and remainder operations, together with associated results. This extension is needed to give correct semantics to Java's integral types in program verification. Special emphasis is put on Java's widening and narrowing functions in relation to the newly defined operations on bitvectors.

1 Introduction

Many programmming languages offer different integral types, represented by different numbers of bits. In Java, for instance, one has integral types `byte` (8 bits), `short` (16 bits), `int` (32 bits) and `long` (64 bits). Additionally, there is a 16 bit type `char` for unicode characters, see [8, §§4.2.1]. It is a usual abstraction in program verification to disregard these differences and interpret all of these types as the unbounded, mathematical integers. However, during the last few years both program verification and theorem proving technology have matured in such a way that more precise representations of integral types can be used.

An important application area for program verification is Java Card based smart cards. Within this setting the above mentioned abstraction of integral types is particularly problematic, because of the following reasons.

- Given the limited resources on a smart card, a programmer chooses his/her integral data types as small as possible, so that potential overflows are a concern (see [6, Chapter 14]). Since such overflows do not produce exceptions in Java (like in Ada), a precise semantics is needed.
- Communication between a smart card and a terminal uses a special structured byte sequence, called an "apdu", see [6]. As a result, many low-level operations with bytes occur frequently, such as bitwise negation or masking.
- Unnoticed overflow may form a security risk: imagine you use a short for a sequence number in a security protocol, which is incremented with every protocol run. An overflow then makes you vulnerable to a possible replay attack.

Attention in the theorem proving community has focused mainly on formalising properties of (IEEE 754) floating-point numbers, see *e.g.* [4, 9, 10, 18]. Such results are of interest in the worlds of microprocessor construction and scientific computation. However, there are legitimate concerns about integral types as well. It is argued in [19] that Java's integral types are unsafe, because overflow is not detected via exceptions, and are confusing because of the asymmetric way that conversions work: arguments are automatically promoted, but results are not automatically "demoted"[1].

Verification tools like LOOP [3] or Krakatoa [14] use the specification language JML [12, 13] in order to express the required correctness properties for Java programs. Similarly, the KeY tool [1] uses UML's Object Constraint Language (OCL). Program properties can also be checked statically with the ESC/Java tool [7], but such checking ignores integral bounds. The theorem prover based approach with the semantics of this paper will take bounds into account. In [5] (see also [2]) it is proposed that a specification language like JML for Java should use the mathematical (unbounded) integers, for describing the results of programs using bounded integral types, because "developers are in a different mindset when reading or writing specifications, particularly when it comes to reasoning about integer arithmetic". This issue is not resolved yet in the program specification community – and it will not be settled here.

Instead, this paper describes the bit-level semantics for Java's integral types developed for the LOOP tool. As such it contributes both to program verification and to library development for theorem provers (esp. for PVS [15]). The semantics is based on PVS's (standard) bitvector library. This PVS library describes bitvectors of arbitrary length, given as a parameter, together with functions bv2nat and bv2int for the unsigned (one's-complement) and signed (two's-complement) interpretation of bitvectors. Associated basic operations are defined, such as addition, subtraction, and concatenation. In this paper, the following items are added to this library.

1. Executable definitions. For instance, the standard library contains "definitions by specification" of the form:

   ```
   -(bv: bvec[N]): { bvn: bvec[N] | bv2int(bvn) =
                       IF bv2int(bv) = minint THEN bv2int(bv)
                       ELSE -(bv2int(bv)) ENDIF}
   *(bv1: bvec[N], bv2: bvec[N]): {bv:bvec[2*N] | bv2nat(bv) =
                                     bv2nat(bv1) * bv2nat(bv2)}
   ```

   Such definitions[2] are not so useful for our program verifications, because sometimes we need to actually compute outcomes. Therefore we give executable redefinitions of these operations. Then we can compute, for instance, `(4*b)&0x0F`.
2. Similarly, executable definitions are introduced for division and remainder operations, which are not present in the standard library. We give such definitions both for unsigned and signed interpretations, following standard hardware realisations via shifting of registers. The associated results are non-trivial challenges in theorem proving.

[1] For a byte (or short) `b`, the assignment `b = b-b` leads to a compile time error: the arguments of the minus function are first converted implicitly to `int`, but the result must be converted explicitly back, as in `b = (byte)(b-b)`.

[2] Readers familiar with PVS will see that these definitions generate so-called type correctness conditions (TCCs), requiring that the above sets are non-empty. These TCCs can be proved via the inverses `int2bv` and `nat2bv` of the (bijective) functions `bv2int` and `bv2nat`, see Section 3. The inverses exist because one has bijections, but they are not executable.

3. Specifically for Java we introduce so-called widening and narrowing, for turning a bitvector of length N into one of length $2 * N$ and back, see [8, §§5.1.2 and §§5.1.3]. When, for example, a byte is added to a short, both arguments are first "promoted" in Java-speak to integers via widening, and then added. Appropriate results are proven relating for instance widening and multiplication or division.

We show how our definitions of multiplication, division and remainder satisfy all properties listed in the Java Language Specification [8, §§15.17.1-3].

In particular we get a good handle on overflow, so that we can prove for the values `minint = 0x80000000` and `maxint = 0x7FFFFFFF`, the truth of the following Java boolean expressions.

```
minint - 1 == maxint       maxint + 1 == minint
minint * -1 == minint      maxint * maxint == 1
minint / -1 == minint
```

As a result the familiar cancellation laws for multiplication ($a * b = a * c \Rightarrow b = c$, for $a \neq 0$) and for division ($\frac{a*b}{a*c} = \frac{b}{c}$, for $a \neq 0, c \neq 0$) do not hold, since:

```
minint * -1 == minint * 1    (minint * -1) / (minint * 1) == 1
```

But these cancellation laws do hold in case there is no overflow. Similarly, we can prove the crucial property of a mask to turn bytes into nonnegative shorts: for a byte `b`,

```
(short)(b & 0xFF) == (b >= 0) ? b : (b + 256)
```

Two more examples are presented in Section 2.

Integral arithmetic is a very basic topic in computer science (see *e.g.* [17]). Most theorem provers have a standard bitvector library that covers the basics, developed mostly for hardware verification. But multiplication, division and remainder are typically not included. The contribution of this paper lies in the logical formalisation of these operations and their results, and in linking the outcome to Java's arithmetic, especially to its widening and narrowing operations. These are the kind of results that "everybody knows" but are hard to find and easy to get wrong.

Of course, one can ask: why go through all this trouble at bitvector level, and why not define the integral operations directly on appropriate bounded intervals of the (mathematical) integers – like for instance in [16] (for the theorem prover Isabelle)? We have several reasons.

– Starting at the lowest level of bits gives more assurance. The non-trivial definitions of the operations that should be used on the bounded intervals appear in our bit-level approach as results about operations that are defined at a lower level (see the final results before Subsection 7.1). This is important, because for instance in [16] it took several iterations (with input from the present approach) to get the correct formulation for the arithmetically non-standard definitions of division and remainer for Java.
– Once appropriate higher-level results are obtained about the bit-level representation, these results can be used for automatic rewriting, without revealing the underlying structure. Hence this approach is at least as powerful as the one with bounded intervals of integers.

– Certain operations from programming languages (like bitwise conjunction or negation) are extremely hard to describe without an underlying bit-level semantics.

This paper has a simple structure. It starts by describing simple Java programs with integral types as motivation. From there it investigates bitvectors. First it explains the basics of PVS's standard bitvector library. Then, in Section 5 it describes our definition of multiplication with associated properties. Division and remainder operations are more difficult; they are first described in unsigned (one's-complement) form in Section 6, and subsequently in signed (two's-complement) form in Section 7. Although the work we have done has been carried out in the language of a specific theorem prover (namely PVS), we shall use a general, mathematical notation to describe it.

2 Java Examples

The following two programs[3] are extremely simple, yet involve some non-trivial semantical issues.

```
int n() {
  for (byte b = Byte.MIN_VALUE;
       b <= Byte.MAX_VALUE; b++) {
         if (b == 0x90) {
               return 1000; };
  }
}
```

```
int s() {
  int n = 0;
  while (-1 << n != 0) {
     n++;
  };
  return n;
}
```

Both these programs hang (*i.e.* loop forever). This can be shown with the LOOP tool, using the integral semantics described in this paper. The reader may wish to pause a moment to understand why these programs hang.

The program `n` on the left hangs because the bound condition `b <= Byte.MAX_VALUE` is always true: the increment operation wraps around. Further, the value `0x90` is interpreted in Java as an integer, and is thus equal to $9 \times 16 = 144$, which is outside the range $[-128, 127]$ used for bytes. Hence the condition of the if-statement is always false, so that one does not return from the loop in this way.

Within the program `s` on the right the integral -1 (which is `0xFFFFFFFF`) is repeatedly shifted to the left. However, Java's leftshift operator << only uses the five lower-order bits of its second argument, see [8, §§15.19]. These five bits can result in a shift of at most $s^5 - 1 = 31$ positions. This is not enough to turn -1 into 0.

This illustrates that a proper understanding of ranges and bitpositions is needed to reason about even elementary Java programs.

3 PVS's Standard Bitvector Library

The distribution of PVS comes with a basic bitvector library[4]. We sketch some ingredients that will be used later. A bit is defined as in PVS as a boolean, but here we shall equivalently use it as an element of $\{0, 1\}$. A bitvector of length N is a

[3] Adapted from `www.linux-mag.com/downloads/2003-03/puzzlers/`.

[4] Developed by Butler, Miner, Carreño (NASA Langley), Miller, Greve (Rockwell Collins) and Srivas (SRI International).

function in $\mathsf{bvec}(N) \overset{\text{def}}{=} \big(\mathsf{below}(N) \to \mathsf{bit}\big)$, where $\mathsf{below}(N)$ is the N-element set $\{0, 1, \ldots, N-1\}$ of natural numbers below N. For instance, the null bitvector is $\lambda i \in \mathsf{below}(N).\, 0$, which we shall often write as $\overrightarrow{0}$, leaving the length N implicit. Similarly, one can write $\overrightarrow{1}$ for $\lambda i \in \mathsf{below}(N).\, 1$. It should be distinguished from **1** = $\lambda i \in \mathsf{below}(N).\, \mathsf{if}\ i = 0\ \mathsf{then}\ 1\ \mathsf{else}\ 0$.

The unsigned interpretation of bitvectors is given by the (parametrised) function $\mathsf{bv2nat}\colon \mathsf{bvec}(N) \to \mathsf{below}(2^N)$, defined as:

$$\begin{aligned} \mathsf{bv2nat}(bv) &\overset{\text{def}}{=} \mathsf{bv2nat\text{-}rec}(bv, N), \quad \text{where} \\ \mathsf{bv2nat\text{-}rec}(bv, n) &\overset{\text{def}}{=} \begin{cases} 0 & \text{if } n = 0 \\ bv(n-1) * 2^{n-1} + \mathsf{bv2nat\text{-}rec}(bv, n-1) & \text{if } n > 0 \end{cases} \end{aligned} \tag{1}$$

Clearly, $\mathsf{bv2nat}$ is bijective. And also: $\mathsf{bv2nat}(bv) = 0 \Leftrightarrow bv = \overrightarrow{0}$, $\mathsf{bv2nat}(bv) = 2^N - 1 \Leftrightarrow bv = \overrightarrow{1}$, and $\mathsf{bv2nat}(bv) = 1 \Leftrightarrow bv = \mathbf{1}$.

The signed interpretation is given by a similar function $\mathsf{bv2int}\colon \mathsf{bvec}(N) \to \{i \in \mathbb{Z} \mid -2^{N-1} \le i \wedge i < 2^{N-1}\}$. It is defined in terms of the unsigned interpretation:

$$\mathsf{bv2int}(bv) \overset{\text{def}}{=} \begin{cases} \mathsf{bv2nat}(bv) & \text{if } \mathsf{bv2nat}(bv) < 2^{N-1} \\ \mathsf{bv2nat}(bv) - 2^N & \text{otherwise.} \end{cases} \tag{2}$$

The condition $\mathsf{bv2nat}(bv) < 2^{N-1}$ means that the most significant bit $bv(N-1)$ is 0. Therefore, this bit is often called the *sign bit*, when the signed interpretation is used. This $\mathsf{bv2int}$ function is also bijective.

The PVS bitvector library provides various basic operations and results. For instance, there is an (executable) addition operation $+$ on bitvectors, introduced via a recursively defined adder. A unary minus operation $-$ is introduced via a specification, as described in the introduction. Binary minus is then defined as: $bv_1 - bv_2 \overset{\text{def}}{=} bv_1 + (-bv_2)$. These operations work for both the unsigned and for the signed interpretation. A typical result is:

$$\begin{aligned} &\mathsf{bv2int}(bv_1 + bv_2) \\ &= \begin{cases} \mathsf{bv2int}(bv_1) + \mathsf{bv2int}(bv_2) & \text{if } -2^{N-1} \le \mathsf{bv2int}(bv_1) + \mathsf{bv2int}(bv_2) \\ & \text{and } \mathsf{bv2int}(bv_1) + \mathsf{bv2int}(bv_2) < 2^{N-1} \\ \mathsf{bv2int}(bv_1) + \mathsf{bv2int}(bv_2) - 2^N & \text{if } \mathsf{bv2int}(bv_1) \ge 0 \text{ and } \mathsf{bv2int}(bv_2) \ge 0 \\ \mathsf{bv2int}(bv_1) + \mathsf{bv2int}(bv_2) + 2^N & \text{otherwise.} \end{cases} \end{aligned}$$

The second cases deals with overflow, and the third one with underflow. The library shows, among other things, that the structure $(\mathsf{bvec}(N), +, \overrightarrow{0}, -)$ is a commutative group.

Also we shall make frequent use of left and right shift operations. For $k \in \mathbb{N}$,

$$\begin{aligned} \mathsf{lsh}(k, bv) &= \lambda i \in \mathsf{below}(N).\ \begin{cases} bv(i-k) & \text{if } i \ge k \\ 0 & \text{otherwise.} \end{cases} \\ \mathsf{rsh}(k, bv) &= \lambda i \in \mathsf{below}(N).\ \begin{cases} bv(i+k) & \text{if } i + k < N \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

4 Widening and Narrowing

As mentioned in the introduction, Java uses so-called widening and narrowing operations to move from one integral type to another. These operations can be described in a parametrised way, as functions:

$$\mathsf{widen}\colon \mathsf{bvec}(N) \to \mathsf{bvec}(2*N) \quad\text{and}\quad \mathsf{narrow}\colon \mathsf{bvec}(2*N) \to \mathsf{bvec}(N)$$

defined as:

$$\begin{aligned} \mathsf{widen}(bv) &\overset{\text{def}}{=} \lambda i \in \mathsf{below}(2*N).\, \begin{cases} bv(i) & \text{if } i < N \\ bv(N-1) & \text{otherwise} \end{cases} \\ \mathsf{narrow}(BV) &\overset{\text{def}}{=} \lambda i \in \mathsf{below}(N).\, BV(i) \end{aligned}$$

Thus, narrowing simply ignores the first N bits. The key property of widening is that the unsigned interpretation is unaffected, in the sense that:

$$\mathsf{bv2int}(\mathsf{widen}(bv)) = \mathsf{bv2int}(bv)$$

A theme that will re-appear several times in this paper is that after widening there is no overflow:

$$\begin{aligned} \mathsf{bv2int}\big(\mathsf{widen}(bv_1) + \mathsf{widen}(bv_2)\big) &= \mathsf{bv2int}(bv_1) + \mathsf{bv2int}(bv_2) \\ \mathsf{bv2int}\big(-\mathsf{widen}(bv)\big) &= -\mathsf{bv2int}(bv). \end{aligned} \tag{3}$$

There are similar results about narrowing. First:

$$\mathsf{narrow}(\mathsf{widen}(bv)) = bv.$$

But also:

$$\begin{aligned} \mathsf{narrow}(BV_1 + BV_2) &= \mathsf{narrow}(BV_1) + \mathsf{narrow}(BV_2) \\ \mathsf{narrow}(-BV) &= -\mathsf{narrow}(BV). \end{aligned}$$

The LOOP tool uses widening and narrowing in the translation of Java's arithmetical expressions. For instance, for a byte `b` and short `s`, a Java expression

```
(short)(b + 2*s)
```

is translated into PVS as:

$$\mathsf{narrow}\Big(\mathsf{widen}(\mathsf{widen}(b)) + 2 * \mathsf{widen}(s)\Big)$$

because the arguments are "promoted" in Java to 32 bit integers before addition and multiplication are applied.

In this way we can explain (and verify in PVS) that for `byte b = -128`, one has in Java: `b-1` is -129 and `(byte)(b-1)` is 127.

5 Multiplication

This section describes bitvector multiplication in PVS, following the standard pen-and-paper approach via repeated shifting and adding. The definition we use works well under both the unsigned and signed interpretation.

In our parametrised setting, we use a recursive definition for multiplication. For two bitvectors $bv_1, bv_2 \colon \mathsf{bvec}(N)$ of length N, we define:

$$bv_1 * bv_2 \stackrel{\text{def}}{=} \mathsf{times\text{-}rec}(bv_1, bv_2, N)$$

where for a natural number n,

$$\begin{aligned}&\mathsf{times\text{-}rec}(bv_1, bv_2, n)\\ &\quad\stackrel{\text{def}}{=} \begin{cases} \overrightarrow{0} & \text{if } n = 0\\ bv_2 + \mathsf{lsh}(1, \mathsf{times\text{-}rec}(\mathsf{rsh}(1, bv_1), bv_2, n-1)) & \text{if } n > 0 \text{ and } bv_1(0) = 1\\ \mathsf{lsh}(1, \mathsf{times\text{-}rec}(\mathsf{rsh}(1, bv_1), bv_2, n-1)) & \text{if } n > 0 \text{ and } bv_1(0) = 0\end{cases}\end{aligned}$$

Note that in this definition $bv_1 * bv_2$ has the same length as bv_1 and bv_2, unlike the multiplication by specification from the standard PVS library (described in the introduction), which doubles the length.

A crucial result is that (our) multiplication can be expressed simply as iterated addition, an appropriate number of times.

$$bv_1 * bv_2 = \mathsf{iterate}(\lambda b \in \mathsf{bvec}(N).\, b + bv_1, \mathsf{bv2nat}(bv_2))(\overrightarrow{0}),$$

where $\mathsf{iterate}(f, k)(x)$ is $f^{(k)}(x) = f(\cdots f(x) \cdots)$, *i.e.* f applied k times to x. This allows us to prove familiar results, like

$$\begin{gathered}\overrightarrow{0} * bv = \overrightarrow{0} \quad bv_1 * bv_2 = bv_2 * bv_1 \quad \mathbf{1} * bv = bv \quad (-bv_1) * bv_2 = -(bv_1 * bv_2)\\ bv_1 * (bv_2 * bv_3) = (bv_1 * bv_2) * bv_3 \qquad bv_1 * (bv_2 + bv_3) = bv_1 * bv_2 + bv_1 * bv_3\end{gathered}$$

They express that $(\mathsf{bvec}(N), *, \mathbf{1})$ is a commutative monoid, and that $*$ preserves the group structure $(\mathsf{bvec}(N), +, \overrightarrow{0}, -)$.

As for the interpretation, the following two results are most relevant.

$$\begin{aligned}&\mathsf{bv2nat}(bv_1) * \mathsf{bv2nat}(bv_2) < 2^N\\ &\qquad\Longrightarrow \mathsf{bv2nat}(bv_1 * bv_2) = \mathsf{bv2nat}(bv_1) * \mathsf{bv2nat}(bv_2)\\ &-2^{N-1} < \mathsf{bv2int}(bv_1) * \mathsf{bv2int}(bv_2) \;\text{ and }\; \mathsf{bv2int}(bv_1) * \mathsf{bv2int}(bv_2) < 2^{N-1}\\ &\qquad\Longrightarrow \mathsf{bv2int}(bv_1 * bv_2) = \mathsf{bv2int}(bv_1) * \mathsf{bv2int}(bv_2).\end{aligned}$$

This means that we have an analogue of (3) for multiplication: after widening there is no overflow:

$$\mathsf{bv2int}(\mathsf{widen}(bv_1) * \mathsf{widen}(bv_2)) = \mathsf{bv2int}(bv_1) * \mathsf{bv2int}(bv_2). \tag{4}$$

This result is very useful in actual calculations (in PVS, about Java programs). For the general situation, with possible over- or under-flow, we have the following formula that slices the (mathematical) integers into appropriate ranges.

$$\begin{aligned} &\forall n \in \mathbb{Z}.\ n * 2^N - 2^{N-1} \leq \mathsf{bv2int}(bv_1) * \mathsf{bv2int}(bv_2) \ \wedge \\ &\qquad \mathsf{bv2int}(bv_1) * \mathsf{bv2int}(bv_2) < n * 2^N + 2^{N-1} \\ &\qquad\quad \Longrightarrow \mathsf{bv2int}(bv_1 * bv_2) = \mathsf{bv2int}(bv_1) * \mathsf{bv2int}(bv_2) - n * 2^N. \end{aligned}$$

Finally, we have the following two results about multiplication and narrowing.

$$\begin{aligned} \mathsf{narrow}(BV_1 * BV_2) &= \mathsf{narrow}(BV_1) * \mathsf{narrow}(BV_2) \\ bv_1 * bv_2 &= \mathsf{narrow}\Big(\mathsf{widen}(bv_1) * \mathsf{widen}(bv_2)\Big) \end{aligned} \tag{5}$$

This second result follows immediately from the first. It is of interest because it shows that our multiplication satisfies the following requirement from the Java Language Specification [8, §§15.17.1]:

> If an integer multiplication overflows, then the result is the low-order bits of the mathematical product as represented in some sufficiently large two's-complement format.

The "lower-order bits" result from the narrow in $\mathsf{narrow}\big(\mathsf{widen}(bv_1) * \mathsf{widen}(bv_2)\big)$ in (5)'s second equation, and the "mathematical product" is its argument $\mathsf{widen}(bv_1) * \mathsf{widen}(bv_2)$, as expressed by (4).

6 Unsigned Division and Remainder

Division and remainder for bitvectors are less straightforward than multiplication. They are based on the same pen-and-paper principles, but the verifications are more involved. In this section we describe a standard machine algorithm for the unsigned interpretation, see *e.g.* [17, 8.3]. The next section adapts this approach to the signed interpretation, and shows how it can be used for division in Java.

In the description below we use arbitrary bitvectors $dvd, dvs, rem, quot, aux$, of a fixed length N. The abbreviation dvd stands for 'dividend', and dvs for 'divisor', to be used in dvd / dvs and dvd % dvs.

Unsigned division and remainder are defined via first and second projections of a recursive auxiliary function:

$$\begin{aligned} \mathsf{div}(dvd, dvs) &\stackrel{\text{def}}{=} \pi_1 \mathsf{divrem}(dvd, dvs, \overrightarrow{0}, N) \\ \mathsf{rem}(dvd, dvs) &\stackrel{\text{def}}{=} \pi_2 \mathsf{divrem}(dvd, dvs, \overrightarrow{0}, N), \end{aligned}$$

where for $n \in \mathbb{N}$,

$$\begin{aligned} \mathsf{divrem}(dvd, dvs, rem, 0) &\stackrel{\text{def}}{=} (dvd, rem) \\ \mathsf{divrem}(dvd, dvs, rem, n+1) &\stackrel{\text{def}}{=} \mathsf{let}\ \ dvd' = \mathsf{lsh}(1, dvd), \\ &\qquad\quad\ rem' = \mathsf{lsh}(1, rem)\ \mathsf{with}\ [(0) := dvd(N-1)] \\ &\qquad \mathsf{in}\ \ \mathsf{if}\ \mathsf{bv2nat}(dvs) \leq \mathsf{bv2nat}(rem') \\ &\qquad\quad\ \mathsf{then}\ \mathsf{divrem}(dvd'\ \mathsf{with}\ [(0) := 1], \\ &\qquad\qquad\qquad dvs, rem' - dvs, n-1) \\ &\qquad\quad\ \mathsf{else}\ \mathsf{divrem}(dvd', dvs, rem', n-1). \end{aligned}$$

The 'with' operator is a convenient short-hand for function update: f with $[(i) := a]$ is the function g with $g(n) = \mathsf{if}\ n = i\ \mathsf{then}\ a\ \mathsf{else}\ f(n)$.

The definition of divrem is quite sneaky and efficient, since it uses very few arguments (or registers in a hardware implementation). The *dvd* argument is shifted from the left into *rem*, and at the same time the quotient result is built up in *dvd* from the right. This overloaded use of *dvd* makes it impossible to formulate an appropriate invariant for this recursive function. Therefore we introduce an alternative function divrem* without this overloading, show that divrem* computes the same result as divrem, and formulate and prove an appropriate invariant for divrem*.

$$
\begin{array}{l}
\mathsf{divrem^*}(aux, dvd, dvs, rem, 0) \stackrel{\text{def}}{=} \\
\qquad (aux, dvd, rem) \\
\mathsf{divrem^*}(aux, dvd, dvs, rem, n+1) \stackrel{\text{def}}{=} \\
\qquad \mathsf{let}\ \ aux' = \mathsf{lsh}(1, aux)\ \mathsf{with}\ [(0) := dvd(N-1)] \\
\qquad\qquad\ \ dvd' = \mathsf{lsh}(1, dvd), \\
\qquad\qquad\ \ rem' = \mathsf{lsh}(1, rem)\ \mathsf{with}\ [(0) := dvd(N-1)] \\
\qquad \mathsf{in}\ \ \mathsf{if}\ \mathsf{bv2nat}(dvs) \le \mathsf{bv2nat}(rem') \\
\qquad\qquad\ \ \mathsf{then}\ \mathsf{divrem^*}(aux', dvd'\ \mathsf{with}\ [(0) := 1], dvs, \\
\qquad\qquad\qquad\qquad \begin{pmatrix} \lambda i \in \mathsf{below}(N).\ \mathsf{if}\ i < N - n + 1 \\ \qquad\qquad\qquad \mathsf{then}\ rem'(i)\ \mathsf{else}\ 0 \end{pmatrix} - dvs, n-1) \\
\qquad\qquad\ \ \mathsf{else}\ \mathsf{divrem^*}(aux', dvd', dvs, rem', n-1).
\end{array}
$$

In this definition the original argument *dvd* is also shifted into the *aux* register, so that it is not lost and can be used for the formulation of the invariant. Also, the subtraction of *dvs* from *rem'*, if possible, happens only from the relevant, lower part of *rem'*.

The fact that divrem and divrem* compute the same results is expressed as follows. For all $n \le N$,

$$
\begin{array}{l}
\mathsf{let}\ \ dr^* = \mathsf{divrem^*}(aux, dvd, dvs, rem, n), \\
\qquad dr = \mathsf{divrem}(dvd, dvs, \begin{pmatrix} \lambda i \in \mathsf{below}(N).\ \mathsf{if}\ i < n \\ \qquad\qquad\qquad \mathsf{then}\ rem(i)\ \mathsf{else}\ 0 \end{pmatrix}, n) \\
\mathsf{in}\ \ \pi_1 dr^* = \lambda i \in \mathsf{below}(N).\ \mathsf{if}\ i < n\ \mathsf{then}\ dvd(N-n+i)\ \mathsf{else}\ aux(i-n)\ \wedge \\
\qquad \pi_2 dr^* = \pi_1 dr\ \wedge \\
\qquad \pi_3 dr^* = \pi_2 dr.
\end{array}
$$

Note that for $n = N$, the third argument of divrem is simply *rem*, and $\pi_1\mathsf{dr^*} = dvd$. As a result, division and remainder can also be expressed in terms of divrem*:

$$
\begin{array}{r}
\mathsf{div}(dvd, dvs) = \pi_2\mathsf{divrem^*}(\overrightarrow{0}, dvd, dvs, \overrightarrow{0}, N) \\
\mathsf{rem}(dvd, dvs) = \pi_3\mathsf{divrem^*}(\overrightarrow{0}, dvd, dvs, \overrightarrow{0}, N).
\end{array}
$$

We are now in a position to express the key invariant property. For $n \leq N$,

$$\begin{array}{rl}
\mathsf{let} & \mathit{dvd\_n} = \lambda i \in \mathsf{below}(N).\, \mathsf{if}\ i < n\ \mathsf{then}\ \mathit{dvd}(N-n+i)\ \mathsf{else}\ \mathit{aux}(i-n), \\
& \mathit{quot\_n} = \lambda i \in \mathsf{below}(N).\, \mathsf{if}\ i < N-n\ \mathsf{then}\ \mathit{dvd}(i)\ \mathsf{else}\ 0, \\
& \mathit{rem\_n} = \lambda i \in \mathsf{below}(N).\, \mathsf{if}\ i < n\ \mathsf{then}\ \mathit{dvd}(N-n+i)\ \mathsf{else}\ \mathit{rem}(i-n), \\
& \mathit{dr*} = \mathsf{divrem*}(\mathit{aux}, \mathit{dvd}, \mathit{dvs}, \mathit{rem}, n) \\
\mathsf{in} & \mathsf{bv2nat}(\mathit{dvd\_n}) = 2^n * \mathsf{bv2nat}(\mathit{dvs}) * \mathsf{bv2nat}(\mathit{quot\_n}) + \mathsf{bv2nat}(\mathit{rem\_n}) \\
& \quad \Longrightarrow \\
& \mathsf{bv2nat}(\pi_1 \mathit{dr*}) = \mathsf{bv2nat}(\mathit{dvs}) * \mathsf{bv2nat}(\pi_2 \mathit{dr*}) + \mathsf{bv2nat}(\pi_3 \mathit{dr*})
\end{array}$$

The proof of this property is far from trivial. The most interesting case is when $n = N$. We then have $\mathit{dvd\_n} = \mathit{dvd}$, $\mathit{quot\_n} = \overrightarrow{0}$ and $\mathit{rem\_n} = \mathit{dvd}$, so that the antecedent of the implication $\Longrightarrow$ trivially holds. This yields a first success:

$$\mathsf{bv2nat}(\mathit{dvd}) = \mathsf{bv2nat}(\mathit{dvs}) * \mathsf{bv2nat}(\mathsf{div}(\mathit{dvd}, \mathit{dvs})) + \mathsf{bv2nat}(\mathsf{rem}(\mathit{dvd}, \mathit{dvs})). \tag{6}$$

It is not hard to prove the expected upperbound for remainder:

$$\mathsf{bv2nat}(\mathit{dvs}) \neq 0 \Longrightarrow \mathsf{bv2nat}(\mathsf{rem}(\mathit{dvd}, \mathit{dvs})) < \mathsf{bv2nat}(\mathit{dvs}). \tag{7}$$

The restriction to non-null divisors is relevant, because:

$$\mathsf{div}(\mathit{dvd}, \overrightarrow{0}) = \overrightarrow{1} \quad \text{and} \quad \mathsf{rem}(\mathit{dvd}, \overrightarrow{0}) = \mathit{dvd}.$$

Division and remainder in Java throw an exception when the divisor is null. This behaviour is realised in the semantics used by the LOOP tool via a wrapper function around the bitvector operations that we are describing. But this wrapper is omitted here.

These two results (6) and (7) characterise division and remainder, in the following sense.

$$\begin{array}{l}
\forall q, r \in \mathbb{N}.\ \mathsf{bv2nat}(\mathit{dvs}) * q + r = \mathsf{bv2nat}(\mathit{dvd}) \wedge r < \mathsf{bv2nat}(\mathit{dvs}) \\
\quad \Longrightarrow q = \mathsf{bv2nat}(\mathsf{div}(\mathit{dvd}, \mathit{dvs})) \wedge r = \mathsf{bv2nat}(\mathsf{rem}(\mathit{dvd}, \mathit{dvs})).
\end{array} \tag{8}$$

This is, together with (6) and (7), the main result of this section. It allows us to prove various results about (unsigned) division and remainder, such as:

$$\mathsf{div}(\mathit{dvd}, \mathbf{1}) = \mathit{dvd} \quad \text{and} \quad \mathsf{rem}(\mathit{dvd}, \mathbf{1}) = \overrightarrow{0}.$$

And:

$$\mathsf{bv2nat}(\mathsf{div}(\mathit{dvd}, \mathit{dvs})) = 0 \Longleftrightarrow \mathsf{bv2nat}(\mathit{dvd}) < \mathsf{bv2nat}(\mathit{dvs}).$$

7 Signed Division and Remainder

Our aim in this section is first to introduce signed division and remainder operations, and prove the analogues of (6), (7) and (8). Next we intend to prove the properties that are listed in the Java Language Specification [8] about division and remainder.

Before we move from unsigned division and remainder to the signed versions (as used in Java), we recall that:

$$\begin{array}{rcl} 5 \mathbin{/} 3 = 1 & \qquad & 5 \mathbin{\%} 3 = 2 \\ 5 \mathbin{/} -3 = -1 & & 5 \mathbin{\%} -3 = 2 \\ -5 \mathbin{/} 3 = -1 & & -5 \mathbin{\%} 3 = -2 \\ -5 \mathbin{/} -3 = 1 & & -5 \mathbin{\%} -3 = -2. \end{array}$$

In line with these results, we make the following case distinctions:

```
dvd / dvs
  ≝ if bv2int(dvd) ≥ 0
    then if bv2int(dvs) ≥ 0
         then div(dvd, dvs)
         else − div(dvd, −dvs))
    else if bv2int(dvs) ≥ 0
         then − div(−dvd, dvs))
         else div(−dvd, −dvs)
```

```
dvd % dvs
  ≝ if bv2int(dvd) ≥ 0
    then if bv2int(dvs) ≥ 0
         then rem(dvd, dvs)
         else rem(dvd, −dvs))
    else if bv2int(dvs) ≥ 0
         then − rem(−dvd, dvs))
         else − rem(−dvd, −dvs)
```

Using the properties of unsigned division and remainder we quickly get:

$$dvd \mathbin{/} \overrightarrow{0} = \begin{cases} \overrightarrow{1} & \text{if } \mathsf{bv2int}(dvd) \geq 0 \\ \mathbf{1} & \text{otherwise.} \end{cases} \qquad dvd \mathbin{\%} \overrightarrow{0} = dvd$$

$$dvd \mathbin{/} \mathbf{1} = dvd \qquad dvd \mathbin{\%} \mathbf{1} = \overrightarrow{0}.$$

The signed analogue of (7) involves the absolute value function:

$$\mathsf{bv2int}(dvs) \neq 0 \Longrightarrow \mathsf{abs}(\mathsf{bv2int}(dvd \mathbin{\%} dvs)) < \mathsf{abs}(\mathsf{bv2int}(dvs)). \tag{9}$$

The analogue of (6) involves an overflow exception:

$$\begin{array}{l} \neg\big(\mathsf{bv2int}(dvd) = -2^{N-1} \wedge \mathsf{bv2int}(dvs) = -1\big) \\ \quad \Longrightarrow \\ \mathsf{bv2int}(dvd) = \mathsf{bv2int}(dvs) * \mathsf{bv2int}(dvd \mathbin{/} dvs) + \mathsf{bv2int}(dvd \mathbin{\%} dvs). \end{array} \tag{10}$$

The proof of this property is obtained from (6), applied after the various case distinctions. The overflow case – when $dvd = \mathsf{minint}$ and $dvs = \overrightarrow{1}$ – does not satisfy (10) because:

$$\mathsf{bv2int}(\mathsf{minint} \mathbin{/} \overrightarrow{1}) = -2^{N-1} \qquad \mathsf{bv2int}(\mathsf{minint} \mathbin{\%} \overrightarrow{1}) = 0.$$

Actually, we can move the bv2int's in (10) to the outside and remove them (because bv2int is injective). This yields:

$$dvd = dvs * \big(dvd \mathbin{/} dvs\big) + \big(dvd \mathbin{\%} dvs\big). \tag{11}$$

The restriction from (10) disappears in this form – where $*$ is multiplication from Section 5, with its own overflow behaviour.

Next we turn to the sign of signed division and remainder. It is most complicated for division.

$$\begin{array}{l}
\neg\big(\mathsf{bv2int}(dvd) = -2^{N-1} \wedge \mathsf{bv2int}(dvs) = -1\big) \;\wedge\; \mathsf{bv2int}(dvs) \neq 0 \\
\qquad \Longrightarrow \\
\Big(\mathsf{bv2int}(dvd \;/\; dvs) > 0 \Leftrightarrow (\mathsf{bv2int}(dvd) \geq \mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) > 0) \\
\qquad\qquad\qquad\qquad\qquad\quad \vee\; (\mathsf{bv2int}(dvd) \leq \mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) < 0)\Big) \\
\wedge \\
\Big(\mathsf{bv2int}(dvd \;/\; dvs) = 0 \Leftrightarrow \mathsf{abs}(\mathsf{bv2int}(dvd)) < \mathsf{abs}(\mathsf{bv2int}(dvs))\Big) \\
\wedge \\
\Big(\mathsf{bv2int}(dvd \;/\; dvs) < 0 \Leftrightarrow (\mathsf{bv2int}(dvd) \geq -\mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) < 0) \\
\qquad\qquad\qquad\qquad\qquad\quad \vee\; (\mathsf{bv2int}(dvd) \leq -\mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) > 0)\Big)
\end{array} \tag{12}$$

About the sign of the remainder we can only say it is determined by the sign of the dividend:

$$\begin{array}{l}
\mathsf{bv2int}(dvd) > 0 \Longrightarrow \mathsf{bv2int}(dvd \;\%\; dvs) \geq 0 \\
\mathsf{bv2int}(dvd) < 0 \Longrightarrow \mathsf{bv2int}(dvd \;\%\; dvs) \leq 0.
\end{array} \tag{13}$$

The uniqueness of signed division and remainder requires more assumptions than in the unsigned case (8). It involves for instance the above sign descriptions.

$$\begin{array}{l}
\forall q, r \in \mathbb{Z}.\; \neg\big(\mathsf{bv2int}(dvd) = -2^{N-1} \wedge \mathsf{bv2int}(dvs) = -1\big) \;\wedge \\
\qquad \mathsf{bv2int}(dvs) \neq 0 \;\wedge \\
\qquad \Big(q > 0 \Leftrightarrow (\mathsf{bv2int}(dvd) \geq \mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) > 0) \\
\qquad\qquad\qquad \vee\; (\mathsf{bv2int}(dvd) \leq \mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) < 0)\Big) \;\wedge \\
\qquad \Big(q = 0 \Leftrightarrow \mathsf{abs}(\mathsf{bv2int}(dvd)) < \mathsf{abs}(\mathsf{bv2int}(dvs))\Big) \;\wedge \\
\qquad \Big(q < 0 \Leftrightarrow (\mathsf{bv2int}(dvd) \geq -\mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) < 0) \\
\qquad\qquad\qquad \vee\; (\mathsf{bv2int}(dvd) \leq -\mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) > 0)\Big) \;\wedge \\
\qquad \Big(\mathsf{bv2int}(dvd) > 0 \Rightarrow r \geq 0\Big) \;\wedge \\
\qquad \Big(\mathsf{bv2int}(dvd) < 0 \Rightarrow r \leq 0\Big) \;\wedge \\
\qquad \mathsf{abs}(r) < \mathsf{abs}(\mathsf{bv2int}(dvs)) \;\wedge \\
\qquad \mathsf{bv2int}(dvs) * q + r = \mathsf{bv2int}(dvd) \\
\qquad\qquad \Longrightarrow \\
\qquad q = \mathsf{bv2int}(dvd \;/\; dvs) \wedge r = \mathsf{bv2int}(dvd \;\%\; dvs).
\end{array} \tag{14}$$

As consequence, we obtain the relation between widening and division & remainder, following (3) for addition and (4) for multiplication.

$$\begin{aligned}
&\neg\big(\mathsf{bv2int}(\mathit{dvd}) = -2^{N-1} \wedge \mathsf{bv2int}(\mathit{dvs}) = -1\big) \\
&\quad\Longrightarrow \mathsf{widen}(\mathit{dvd}) \mathbin{/} \mathsf{widen}(\mathit{dvs}) = \mathsf{widen}(\mathit{dvd} \mathbin{/} \mathit{dvs}) \wedge \\
&\quad\ \mathsf{widen}(\mathit{dvd}) \mathbin{\%} \mathsf{widen}(\mathit{dvs}) = \mathsf{widen}(\mathit{dvd} \mathbin{\%} \mathit{dvs}).
\end{aligned} \tag{15}$$

As another consequence we can relate division and remainder for bitvectors to their mathematical counterparts (for integers). In order to do so we use the floor and fractional functions[5] from the standard PVS library. For a real x, $\mathsf{floor}(x)$ is the unique integer i with $i \leq x < i+1$. And $\mathsf{fractional}(x)$ is then $x - \mathsf{floor}(x)$, which is in the interval $[0,1)$. The main result is split in two parts. The difference is in the "$+1$" and "-1" in the last two lines.

$$\begin{aligned}
&\neg\big(\mathsf{bv2int}(\mathit{dvd}) = -2^{N-1} \wedge \mathsf{bv2int}(\mathit{dvs}) = -1\big) \ \wedge \\
&\quad\big((\mathsf{bv2int}(\mathit{dvd}) = 0 \wedge \mathsf{bv2int}(\mathit{dvs}) \neq 0) \ \vee \\
&\quad\ (\mathsf{bv2int}(\mathit{dvd}) > 0 \wedge \mathsf{bv2int}(\mathit{dvs}) > 0) \ \vee \\
&\quad\ (\mathsf{bv2int}(\mathit{dvd}) < 0 \wedge \mathsf{bv2int}(\mathit{dvs}) < 0) \ \vee \\
&\quad\ (\mathsf{bv2int}(\mathit{dvs}) \neq 0 \wedge \exists n \in \mathbb{Z}.\, \mathsf{bv2int}(\mathit{dvd}) = n * \mathsf{bv2int}(\mathit{dvs}))\ \big) \\
&\Longrightarrow \mathsf{bv2int}(\mathit{dvd} \mathbin{/} \mathit{dvs}) = \mathsf{floor}(\mathsf{bv2int}(\mathit{dvd}) \mathbin{/} \mathsf{bv2int}(\mathit{dvs})) \ \wedge \\
&\quad\ \mathsf{bv2int}(\mathit{dvd} \mathbin{\%} \mathit{dvs}) = \mathsf{bv2int}(\mathit{dvs}) * \mathsf{fractional}(\mathsf{bv2int}(\mathit{dvd}) \mathbin{/} \mathsf{bv2int}(\mathit{dvs})).
\end{aligned}$$

$$\begin{aligned}
&\big((\mathsf{bv2int}(\mathit{dvd}) > 0 \wedge \mathsf{bv2int}(\mathit{dvs}) < 0) \ \vee \\
&\ (\mathsf{bv2int}(\mathit{dvd}) < 0 \wedge \mathsf{bv2int}(\mathit{dvs}) > 0)\ \big) \ \wedge \\
&\neg\exists n \in \mathbb{Z}.\, \mathsf{bv2int}(\mathit{dvd}) = n * \mathsf{bv2int}(\mathit{dvs})) \\
&\Longrightarrow \mathsf{bv2int}(\mathit{dvd} \mathbin{/} \mathit{dvs}) = \mathsf{floor}(\mathsf{bv2int}(\mathit{dvd}) \mathbin{/} \mathsf{bv2int}(\mathit{dvs})) + 1 \ \wedge \\
&\quad\ \mathsf{bv2int}(\mathit{dvd} \mathbin{\%} \mathit{dvs}) = \mathsf{bv2int}(\mathit{dvs}) * \big(\mathsf{fractional}(\mathsf{bv2int}(\mathit{dvd}) \mathbin{/} \mathsf{bv2int}(\mathit{dvs})) - 1\big).
\end{aligned}$$

Such results are used as definitions in [16].

7.1 Division in Java

We start with a quote from the Java Language Specification [8, §§15.17.2].

> Integer division rounds toward 0. That is, the quotient produced for operands n and d that are integers after binary numeric promotion (§5.6.2) is an integer value q whose magnitude is as large as possible while satisfying $|d * q| \leq |n|$; moreover, q is positive when and n and d have the same sign, but q is negative when and n and d have opposite signs. There is one special case that does not satisfy this rule: if the dividend is the negative integer of largest possible magnitude for its type, and the divisor is -1, then integer overflow occurs and the result is equal to the dividend. Despite the overflow, no exception is thrown in this case. On the other hand, if the value of the divisor in an integer division is 0, then an ArithmeticException is thrown.

[5] Developed by Paul Miner.

We check that all these properties hold for our signed division and remainder operations defined on bitvectors in PVS. The first property stating that the quotient is "…as large as possible …" is formalised (and proven) as:

$$\begin{aligned}\forall q\colon \mathbb{Z}.\ \mathsf{bv2int}(\mathit{dvs}) \neq 0 \wedge \mathsf{abs}(\mathsf{bv2int}(\mathit{dvs}) * q) \leq \mathsf{abs}(\mathsf{bv2int}(\mathit{dvd})) \\ \Longrightarrow \mathsf{abs}(q) \leq \mathsf{abs}(\mathsf{bv2int}(\mathit{dvd}\ /\ \mathit{dvs})).\end{aligned}$$

The sign of the quotient has already been described in (12). And the "…one special case …" in this quote refers to the assumption in (10).

7.2 Remainder in Java

The relevant quote [8, §§15.17.3] says:

> The remainder operation for operands that are integers after binary numeric promotion (§5.6.2) produces a result value such that $(a/b) * b + (a\%b)$ is equal to a. This identity holds even in the special case that the dividend is the negative integer of largest possible magnitude for its type and the divisor is -1 (the remainder is 0). It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative, and can be positive only if the dividend is positive; moreover, the magnitude of the result is always less than the magnitude of the divisor. If the value of the divisor for an integer remainder operator is 0, then an ArithmeticException is thrown.

The identity "$(a/b) * b + (a\%b)$ is equal to a" in this quote holds as (11), indeed without the restriction that occurs in (10). The statement about the sign of the remainder is stated in (13), and about its magnitude in (9).

We conclude that all properties of division and remainder required in the Java Language Specification hold for our formalisation in PVS.

8 Conclusions

This paper formalises the details of multiplication, division and remainder operations for bitvectors in the higher order logic of the theorem prover PVS, and makes precise which properties that this formalisation satisfies. This is typical theorem prover work, involving many subtle details and case distinctions (which humans easily get wrong). The main application area is Java program verification. Therefore, the relation between the newly defined bitvector operations and Java's widening and narrowing functions gets much attention.

The theories underlying this paper have recently been included (by Sam Owre) in the bitvector library of PVS version 3.0 (and upwards). Also, the bitvector semantics is now heavily used for verifying specific Java programs, see for instance [11].

Acknowledgements

Thanks are due to Joseph Kiniry and Erik Poll for their feedback on this paper.

References

1. W. Ahrendt, Th. Baar, B. Beckert, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, and P.H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In R.-D. Knutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE)*, number 2306 in Lect. Notes Comp. Sci., pages 327–330. Springer, Berlin, 2002.
2. B. Beckert and S. Schlager. Integer arithmetic in the specification and verification of Java programs. Workshop on Tools for System Design and Verification (FM-TOOLS), `http://i12www.ira.uka.de/~beckert/misc.html`, 2002.
3. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in Lect. Notes Comp. Sci., pages 299–312. Springer, Berlin, 2001.
4. V.A. Carreño and P.S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In E.Th. Schubert, Ph.J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications*, 1995. Category B Proceedings.
5. P. Chalin. Improving jml: For a safer and more effective language. In *Formal Methods Europe 2003*, 2003, to appear.
6. Z. Chen. *Java Card Technology for Smart Cards*. The Java Series. Addison-Wesley, 2000.
7. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37(5) of *SIGPLAN Notices*, pages 234–245. ACM, 2002.
8. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, 2000.
9. J. Harrison. A machine-checked theory of floating point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics*, number 1690 in Lect. Notes Comp. Sci., pages 113–130. Springer, Berlin, 1999.
10. J. Harrison. Formal verification of IA-64 division algorithms. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, number 1869 in Lect. Notes Comp. Sci., pages 234–251. Springer, Berlin, 2000.
11. B. Jacobs, M. Oostdijk, and M. Warnier. Formal verification of a secure payment applet. *Journ. of Logic and Algebraic Programming*, To appear.
12. 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.
13. G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, and C. Ruby. JML reference manual (draft). `www.jmlspecs.org`, 2003.
14. C. Marché, C. Paulin, and X. Urbain. The Krakatoa tool for certification java/javacard programs annotated in jml. *Journ. of Logic and Algebraic Programming*, To appear.
15. S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.
16. N. Rauch and B. Wolff. Formalizing Java's two's-complement integral type in Isabelle/HOL. In Th. Arts and W. Fokkink, editors, *Formal Methods for Industrial Critical Systems (FMICS'03)*, number 80 in Elect. Notes in Theor. Comp. Sci. Elsevier, Amsterdam, 2003. `www.elsevier.nl/locate/entcs/volume80.html`.
17. W. Stallings. *Computer Organization and Architecture*. Prentice Hall, 4th edition, 1996.
18. L. Théry. A library for floating-point numbers in Coq. `www-sop.inria.fr/lemme/AOC/coq/`, 2002.
19. J.F.H. Winkler. A safe variant of the unsafe integer arithmetic of Java$^{\mathrm{TM}}$. *Software – Practice and Experience*, 33:669–701, 2002.

Towards Object-Oriented Graphs and Grammars

Ana Paula Lüdtke Ferreira[1] and Leila Ribeiro[2]

[1] Centro de Ciências Exatas e Tecnológicas, Universidade do Vale do Rio dos Sinos
anapaula@exatas.unisinos.br

[2] Instituto de Informática, Universidade Federal do Rio Grande do Sul
leila@inf.ufrgs.br

Abstract. This work aims to extend the algebraical approach to graph transformation to model object-oriented systems structures and computations. A graph grammar based formal framework for object-oriented system modeling is presented, incorporating both the static aspects of system modeling and the dynamic aspects of computation of object-oriented programs.

1 Introduction

Graphs are a very natural way of describing complex situations on an intuitive level. Graph-based formal description techniques are, for that reason, easily used by non-specialists on formal methods. Graph transformation rules can be brought into those descriptions in order to enrich them with a dynamical behaviour, by modeling the evolution of the structures represented as graphs.

The algebraic approach to graph grammars has been presented for the first time in [5] in order to generalize Chomsky grammars from strings to graphs. That approach is currently known as *double-pushout* approach, because derivations are based on two pushout constructions in the category of graphs and total graph morphisms. The *single-pushout* approach, on the other hand, has derivations characterized as a pushout construction in the category of graphs and partial graph morphisms. As presented in [4], [9] and [12], this approach is particularly adequate to model parallelism and distribution.

Graph grammars are appealing as a specification formalism because they are formal, they are based on simple yet powerful concepts to describe behaviour, they have a nice graphical layout which helps the understanding (even by non-specialists in formal methods) of a specification. Since graph grammars also provide a model of computation [7], they can serve as the basis for specifications which can be executed on a computer.

Different kinds of graph grammars have been proposed in the literature [11, 10, 9, 7, 1], aiming the solution of different problems. However, those focusing on object-oriented systems specification [3, 8] do not present a treatment on inheritance and polymorphism, which make object-oriented systems analysis and testing so difficult.

The use of the object-oriented paradigm has increased over the past years, becoming perhaps the most popular paradigm of system development in use nowadays. The growing use of Java as a language to support Internet applications has also contributed to this popularity. Object-based systems have a number of advantages over traditional ones, such as ease of specification, code reuse, modular development and implementation independence. However, they also present difficulties, derived from the very same features that allow the mentioned advantages.

The most distinguished features of object-oriented systems are inheritance and polymorphism, which make them considerably different from other systems in both their architecture and model of execution. It should be expected that formalisms for the specification of object-oriented architectures or programs reflect these particularities, otherwise the use of such formalisms will neglect key concepts that have a major influence in their organization. According to [6], a specification language for object-oriented conceptual modeling must at least include constructs for specifying primitive objects, particularizations of predefined objects, inheritance relationships between objects and aggregation of objects in order to define more complex objects. We also believe that the concepts of polymorphism and dynamic binding are essential if we intend to model static *and* dynamic aspects of object-oriented systems. So, in order to correctly model object-oriented systems, the key concepts related to it must be present within the formalism used.

This work aims to extend the algebraical approach to graph transformations to model object-oriented systems structures and computations. More accurately, the single pushout approach in the category of typed hypergraphs and partial typed hypergraph morphisms will be adapted to fit more adequately the object-oriented approach to software development. We will also show how the structures developed along the text are compatible with the notion of specification and computation within the object-oriented paradigm.

This paper is organized as follows: Section 2 presents a number of special kinds of (hyper)graphs, from where the main concepts of objects, attributes, methods and inheritance are developed. Section 3 presents the fundamental notions of object-oriented graph productions, grammars and derivations. Some examples of productions and its consequences on graph derivations are portrayed. Finally, Section 4 presents some conclusions from the work presented here.

2 Object-Oriented Graphs

The general definition of graphs is specialized to deal with the object-oriented aspects of program specification. This specialization is meant to reflect more precisely the underlying structure of the object-oriented paradigm, and so improve the compactness and understandability of specifications. Object-oriented systems consist of instances of previously defined classes (or objects[1]) which have an internal structure defined by attributes and communicate among themselves

[1] Object-based and class-based models are actually equivalent in terms of what they can represent and compute [13], so we will use the more commonly used class-based approach.

solely through message passing, so that approach underlies the structure of the graphs used to model those systems. Such structures are called *object-model graphs* and their formal definition is given next.

Definition 1 (Object-model graph). *An* object-model graph $\mathcal{M}$ *is a labelled hypergraph* $\langle V_{\mathcal{M}}, E_{\mathcal{M}}, L_{\mathcal{M}}, src_{\mathcal{M}}, tar_{\mathcal{M}}, lab_{\mathcal{M}}\rangle$ *where* $V_{\mathcal{M}}$ *is a finite set of vertices,* $E_{\mathcal{M}}$ *is a finite set of hyperarcs,* $L_{\mathcal{M}} = \{\text{attr}, \text{msg}\}$ *is the set of hyperarcs labels,* $src_{\mathcal{M}}, tar_{\mathcal{M}} : E_{\mathcal{M}} \to V_{\mathcal{M}}^*$ *are the hyperarcs source and target functions,* $lab_{\mathcal{M}} : E_{\mathcal{M}} \to L_{\mathcal{M}}$ *is the hyperarcs labelling function and, for all* $e \in E_{\mathcal{M}}$*, the following holds:*

- *if* $lab_{\mathcal{M}}(e) = \text{attr}$ *then* $src_{\mathcal{M}}(e) \in V_{\mathcal{M}}$ *and* $tar_{\mathcal{M}}(e) \in V_{\mathcal{M}}^*$*, and*
- *if* $lab_{\mathcal{M}}(e) = \text{msg}$ *then* $src_{\mathcal{M}}(e) \in V_{\mathcal{M}}^*$ *and* $tar_{\mathcal{M}}(e) \in V_{\mathcal{M}}$*.*

Sets $\{e \in E_{\mathcal{M}} \mid lab_{\mathcal{M}}(e) = \text{attr}\}$ *and* $\{e \in E_{\mathcal{M}} \mid lab_{\mathcal{M}}(e) = \text{msg}\}$ *are denoted by* $E_{\mathcal{M}}|_{\text{attr}}$ *and* $E_{\mathcal{M}}|_{\text{msg}}$*, respectively.*

Object-model graphs can also be viewed as a definition of the *classes* belonging to a system, where each node is a class identifier, hyperarcs departing from it correspond to its internal attributes, and messages addressed to it consist on the services it provides to the exterior (i.e., its methods). Notice that the restrictions put to the structure of the hyperarcs assure, as expected, that messages target and attributes belong to a single object.

A key feature of the object-oriented paradigm is the notion of *inheritance.* Inheritance is the construction which permits an object to be specialized from a pre-existing one. The newly created object carries all the functionality from its primitive object. This relation induces a hierarchical relationship among the objects from a system, which can be viewed as a set of trees (single inheritance) or as an acyclic graph (multiple inheritance). Both structures can be formally characterized as strict order relations, as follows.

Definition 2 (Strict order relation). *A binary relation* $R \subseteq A \times A$ *is said a* strict order relation *if and only if it has the following properties:*

1. *if* $(a, a') \in R$ *then* $a \neq a'$ *(R has no reflexive pairs);*
2. *if* $(a, a_1), (a_1, a_2), \ldots, (a_{n-1}, a_n), (a_n, a') \in R$*,* $n \geqslant 0$*, then* $(a', a) \notin R$ *(R has no cycles);*
3. *for any* $a, a', a'' \in A$*, if* $(a, a'), (a, a'') \in R$ *then* $a' = a''$ *(R is a function).*

Definition 3 (Type hierarchy). *Let* $\mathcal{M} = \langle V_{\mathcal{M}}, E_{\mathcal{M}}, L_{\mathcal{M}}, src_{\mathcal{M}}, tar_{\mathcal{M}}, lab_{\mathcal{M}}\rangle$ *be an object-model graph. A* type hierarchy *over* $\mathcal{M}$ *is a tuple* $\mathcal{H}_{\mathcal{M}} = \langle \mathcal{M}, isa, redef\rangle$*, where* $isa \subseteq V_{\mathcal{M}} \times V_{\mathcal{M}}$ *and* $redef \subseteq E_{\mathcal{M}} \times E_{\mathcal{M}}$ *are strict order relations*[2] *holding the following properties:*

1. *for each* $(e, e') \in redef$*,* $lab_{\mathcal{M}}(e) = lab_{\mathcal{M}}(e') = \text{msg}$*,*
2. *for each* $(e, e') \in redef$*,* $src_{\mathcal{M}}(e) = src_{\mathcal{M}}(e')$*,*

[2] For any binary relation $r \subseteq A \times A$, r^+ will denote its transitive closure and r^* will denote its reflexive and transitive closure.

3. *for each* $(e, e') \in redef$, $(tar_{\mathcal{M}}(e), tar_{\mathcal{M}}(e')) \in isa^+$, *and*
4. *for each* $(e', e), (e'', e) \in redef$, *if* $e' \neq e''$ *then* $(tar(e'), tar(e'')) \notin isa^*$ *and* $(tar(e''), tar(e')) \notin isa^*$.

The purpose of the relation isa is to establish an inheritance relation between objects. Notice that only single inheritance is allowed, since both isa and $redef$ are required to be functions. Function $redef$ establishes which methods will be redefined within the derived object, by mapping them. The restrictions applied to function $redef$ ensure that methods are redefined consistently, i.e., only two message arcs can be mapped (1), their parameters are the same (2), the method being redefined is located somewhere (strictly) above in the type hierarchy (under isa^+) (3), and only the closest message with respect to relations isa and $redef$ can be redefined (4).

Notice that the requirement concerning the acyclicity and the non reflexivity on isa and $redef$ is consistent with the definition of classes in the object-oriented paradigm. A class is created as a specialization of at most one other class (single inheritance), which must exist prior to the creation of the new class.

Remark 1. Since type hierarchies are algebraic structures, operations over them can be defined. *Composition* (done with or without identification of elements on the structures being composed) plays an important role, since it corresponds to system composition. Although composition is an extremely relevant feature for system development, it is beyond the scope of this article.

Definition 4 (Hierarchy compatible strings). *Given a type hierarchy* $\mathcal{H}_{\mathcal{M}} = \langle \mathcal{M}, isa, redef \rangle$, *two node strings* $u, v \in V_{\mathcal{M}}^*$ *are* hierarchy compatible *if and only if* $|u| = |v|$ *and* $(u_i, v_i) \in isa^*$, $i = 1, \ldots, |u|$. *If* u *and* v *are hierarchy compatible we write* $u \propto_{\mathcal{H}} v$.

The definition of hierarchy compatible strings extends the relation isa^* to strings, which must have the same length, and the corresponding elements must be connected within that relation. It is easy to see that both isa^* and $\propto_{\mathcal{H}}$ are partial order relations.

Example 1. Fig. 1 presents a (naïve) type hierarchy for geometric shapes and figures. The nodes in the graph denote objects (shape, round, circle, ellipse, Figure, Drawing, Color and Integer), while object attributes and messages are represented by hyperarcs. The inheritance relation isa is represented by dotted arrows and the redefinition function $redef$ is represented by solid thin ones.

Definition 5 (Hierarchical graph). *A hierarchical graph* $G^{\mathcal{H}}$ *is a tuple* $\langle G, t, \mathcal{H}_{\mathcal{M}} \rangle$, *where* $\mathcal{H}_{\mathcal{M}} = \langle \mathcal{M}, isa, redef \rangle$ *is a type hierarchy,* G *is a hypergraph, and* t *is a pair of total functions* $\langle t_V : V_G \to V_{\mathcal{M}}, t_E : E_G \to E_{\mathcal{M}} \rangle$ *such that* $(t_V \circ src_G) \propto_{\mathcal{H}} (src_{\mathcal{M}} \circ t_E)$, *and* $(t_V \circ tar_G) \propto_{\mathcal{H}} (tar_{\mathcal{M}} \circ t_E)$.

Hierarchical graphs are hypergraphs typed over an object-model graph which carries a hierarchical relation among its nodes. Notice that the typing morphism is slightly different from the traditional one [12]: a hierarchical graph arc can

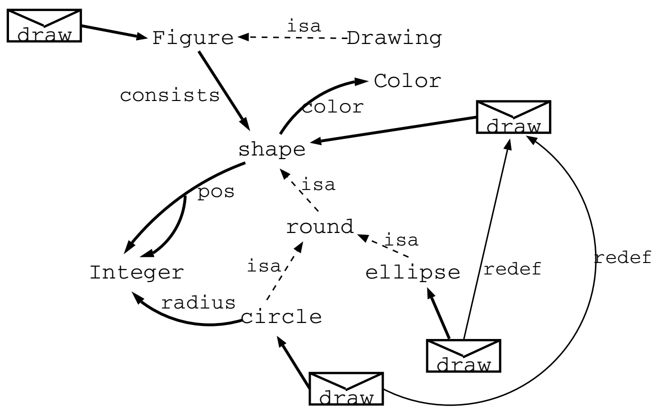


Fig. 1. Type hierarchy for geometric figures

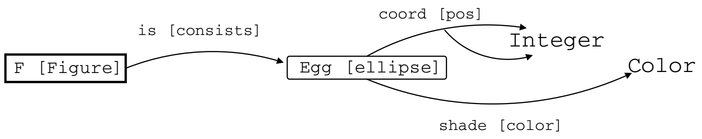


Fig. 2. Example of a hierarchical graph

be incident to any string of nodes which is hierarchy compatible to the one connected to its typing edge. This definition reflects the idea that an object can use any attribute one of its primitive classes have, since it was inherited when the class was specialized.

Example 2. Fig. 2 shows a hierarchical graph typed over the type hierarchy portrayed in Fig. 1. The typing morphism is revealed by the names between brackets. Notice that an ellipse has no attribute directly connected to it in the object-model graph. However, since an *ellipse* is a specialized *shape*, it inherits all its attributes.

Notice that all attributes belonging to the hierarchical graph on Fig. 2 are allowed by Definition 5. The referred graph has three edges, namely *is* (typed as *consists*), *coord* (typed as *pos*), and *shade* (typed as *color*). For *coord* we have (the same can be done to the other two):

$$(t_V \circ src)(coord) = ellipse \propto_{\mathcal{H}} shape = (src_{\mathcal{M}} \circ t_E)$$
$$(t_V \circ tar)(coord) = Integer\,Integer \propto_{\mathcal{H}} Integer\,Integer = (tar_{\mathcal{M}} \circ t_E)$$

Remark 2. For all diagrams presented in the rest of this text, $\rightarrowtail$-arrows denote total morphisms whereas $\rightarrow$-arrows denote arbitrary morphisms (possibly partial). For a partial function f, $dom(f)$ represents its domain of definition, $f?$ and $f!$ denote the corresponding domain inclusion and domain restriction. Each morphism f within category **SetP** can be factorized into components $f?$ and $g!$.

Definition 6 (Hierarchical graph morphism). *Let $G_1^{\mathcal{H}} = \langle G_1, t_1, \mathcal{H}_{\mathcal{M}} \rangle$ and $G_2^{\mathcal{H}} = \langle G_2, t_2, \mathcal{H}_{\mathcal{M}} \rangle$ be two hierarchical graphs typed over the same type hierarchy $\mathcal{H}_{\mathcal{M}}$. A* hierarchical graph morphism *$h : G_1^{\mathcal{H}} \to G_2^{\mathcal{H}}$ between $G_1^{\mathcal{H}}$ and $G_2^{\mathcal{H}}$, is a pair of partial functions $h = \langle h_V : V_{G_1} \to V_{G_2}, h_E : E_{G_1} \to E_{G_2} \rangle$ such that the diagram (in **SetP**)*

$$\begin{tikzcd}
\{\text{attr}, \text{msg}\} \arrow[rr, leftrightarrow, "id_{\{\text{attr},\text{msg}\}}"] & & \{\text{attr}, \text{msg}\} \\
E_{G_1} \arrow[u, "lab_{\mathcal{M}} \circ t_{1E}"', |->] \arrow[d, "src_{G_1}, tar_{G_1}"', |->] & dom(h_E) \arrow[l, "h_E?"', |->] \arrow[r, "h_E!", |->] & E_{G_2} \arrow[u, "lab_{\mathcal{M}} \circ t_{2E}", |->] \arrow[d, "src_{G_2}, tar_{G_2}", |->] \\
V_{G_1}^{*} \arrow[rr, "h_V^{*}"] & & V_{G_2}^{*}
\end{tikzcd}$$

commutes, for all elements $v \in dom(h_V)$, $(t_{2V} \circ h_V)(v) \propto_{\mathcal{H}} t_{1V}(v)$, and for all elements $e \in dom(h_E)$, $((t_{2E} \circ h_E)(e), t_{1E}(e)) \in redef^{*}$. If $(t_{2E} \circ h_E)(e) = t_{1E}(e)$ for all elements $e \in dom(h_E)$, the morphism is said to be strict.

A graph morphism is a mapping which preserves hyperarcs origins and targets. Ordinary typed graph morphisms, however, cannot describe correctly morphisms on object-oriented systems because the existing inheritance relation among objects causes that manipulations defined for objects of a certain kind are valid to all objects derived from it. So, an object can be viewed as not being uniquely typed, but having a type *set* (the set of all types it is connected via isa^{*}).

The meaning of the connection of two elements $x \dashrightarrow y$ by the relation *isa* is the usual: in any place that an object of type y is expected, an object of type[3] x can appear, since an object of type x is also an object of type y. A hierarchical graph morphism reflects this situation, since two nodes can be identified by the morphism as long as they are connected in the reflexive and transitive closure of *isa* within the type hierarchy. Similarly, two arcs can be identified by a hierarchical graph morphism if their types are related by the method redefinition relation. Since attribute arcs are only related (under $redef^{*}$) to themselves, two of them can only be identified if they have the *same* type in the underlying object-model graph. A message, however, can be identified with any other message which redefines it. The reason for this will be clear in Section 3.

It should be noticed that the arity of methods is preserved by the morphism, since two hyperarcs can only be mapped if they have the same number of parameters with compatible types.

[3] The word *type* is used here in a less strict sense than it is used in programming language design texts. Although the literature makes a difference on subtyping and inheritance relationships between objects [2], such differentiation will not be made here, since this work is being done in an upper level of abstraction. It is hoped that this will not cause any confusion to the reader.

Lemma 1. *Hierarchical graph morphisms are closed under composition.*

Proof. Hierarchical graph morphisms are componentwise composable. Given two hierarchical graph morphisms $f = \langle f_V, f_E \rangle : G_1^{\mathcal{H}} \to G_2^{\mathcal{H}}$ and $h = \langle h_V, h_E \rangle : G_2^{\mathcal{H}} \to G_3^{\mathcal{H}}$, the compound morphism $h \circ f = \langle h_V \circ f_V, h_E \circ f_E \rangle : G_1^{\mathcal{H}} \to G_3^{\mathcal{H}}$ exists, since morphisms on **SetP** are composable.
Additionally, for all elements $v \in dom(f_V)$, $(t_{2V} \circ f_V)(v) \propto_{\mathcal{H}} t_{1V}(v)$, and for all elements $v \in dom(h_V)$, $(t_{3V} \circ h_V)(v) \propto_{\mathcal{H}} t_{2V}(v)$ (f and g are both hierarchical graph morphisms). Then, for all $v_1 \in V_{G_1}$, $(t_{2V} \circ f_V)(v_1) \propto_{\mathcal{H}} t_{1V}(v_1)$, and for all $f_V(v_1) \in V_{G_2}$, $(t_{3V} \circ h_V \circ f_V)(v_1) \propto_{\mathcal{H}} (t_{2V} \circ f_V)(v_1)$. Since $\propto_{\mathcal{H}}$ is transitive, $(t_{3V} \circ h_V \circ f_V)(v_1) \propto_{\mathcal{H}} t_{1V}(v_1)$. Thus, $h \circ f$ is a hierarchical graph morphism. □

Lemma 2. *Composition of hierarchical graph morphisms is associative.*

Proof. Composition of hierarchical graph morphisms is done componentwise, and each of the components are functions. Since composition of partial functions and the transitivity of binary relations are associative, so is the composition of hierarchical graph morphisms. □

Proposition 1 (Category GraphP($\mathcal{H}_{\mathcal{M}}$)). *There is a category* **GraphP** *($\mathcal{H}_{\mathcal{M}}$) which has hierarchical graphs over a type hierarchy $\mathcal{H}_{\mathcal{M}}$ as objects and hierarchical graph morphisms as arrows.*

Proof. Lemma 1 proves that the composition of two hierarchical graph morphisms is a hierarchical graph morphism. Lemma 2 states that composition of hierarchical graph morphisms is associative.

The identity morphism for a given hierarchical graph G is the trivial morphism $id_G = \langle id_V, id_E \rangle : G \to G$, where for any vertex $v \in V_G$, $id_{GV}(v) = v$, and for any edge $e \in E_G$, $id_{GE}(e) = e$. So, given any hierarchical graph morphism $h = \langle h_V, h_E \rangle : G_1^{\mathcal{H}} \to G_2^{\mathcal{H}}$, for any vertex $v \in V_{G_1}$, $(h_V \circ id_{G_1V})(v) = h_V(id_{G_1V}(v)) = h_V(v) = id_{G_2V}(h_V(v)) = (id_{G_2V} \circ h_V)(v)$. Similarly, for any edge $e \in E_{G_1}$, $(h_E \circ id_{G_1E})(e) = h_E(id_{G_1E}(e)) = h_E(e) = id_{G_2E}(h_E(e)) = (id_{G_2E} \circ h_E)(e)$.

The existence of identity, composition, and associativity of composition proves that **GraphP**($\mathcal{H}_{\mathcal{M}}$) is a category. □

Since **GraphP**($\mathcal{H}_{\mathcal{M}}$) is a category, the existence of categorical constructs within it can be investigated. However, the general definition of hierarchical graphs must be narrowed to correctly represent object-oriented systems. To achieve this goal, some additional functions and structures will be defined next.

Definition 7 (Attribute set function). *Given a hierarchical graph $\langle G, t, \mathcal{H}_{\mathcal{M}} \rangle$, where $\langle \mathcal{M}, isa, redef \rangle$ is a type hierarchy, let the* attribute set function *$attr_G : V_G \to 2^{E_G}$ return for each vertex $v \in V_G$ the set $\{e \in E_G \mid src_G(e) = v \wedge lab_{\mathcal{M}}(t(e)) = \mathrm{attr}\}$.*

The extended attribute set function, *$attr^*_{\mathcal{M}} : V_{\mathcal{M}} \to 2^{E_{\mathcal{M}}}$, returns for each vertex $v \in V_{\mathcal{M}}$ the set $\{e \in E_{\mathcal{M}} \mid lab_{\mathcal{M}}(e) = \mathrm{attr} \wedge src_{\mathcal{M}}(e) = v' \wedge (v, v') \in isa^*\}$.*

The *attribute set function* and the *extended attribute set function* will help define some other functions, relations and structures along this text. Basically, for any vertex v of a hierarchical graph, the attribute set function returns the set of all attribute arcs having v as their source. Similarly, given a type hierarchy, and a vertex v of its object-model graph, the extended attribute set function returns the set of all attribute arcs whose source is v or any other vertex to which v connected via isa^*.

Definition 8 (Message set function). *Given a hierarchical graph $\langle G, t, \mathcal{H}_\mathcal{M} \rangle$, where $\langle \mathcal{M}, isa, redef \rangle$ is a type hierarchy, let the* message set function $msg_G : V_G \rightarrow 2^{E_G}$ *returns for each vertex $v \in V_G$ the set $\{e \in E_G \mid tar_G(e) = v \wedge lab_\mathcal{M}(t_E(e)) = \text{msg}\}$.*

The extended message set function*, $msg^*_\mathcal{M} : V_\mathcal{M} \rightarrow 2^{E_\mathcal{M}}$, returns for each vertex $v \in V_\mathcal{M}$ the set $\{e \in E_\mathcal{M}|_\text{msg} \mid tar_\mathcal{M}(e) = v' \wedge (v, v') \in isa^* \wedge \forall (e' \neq e) \in E_\mathcal{M}|_\text{msg}((v, tar(e')) \in isa^*, (tar(e'), v') \in isa^* \rightarrow (e', e) \notin redef^*)\}$.*

The *message set function* returns all messages an object within an hierarchical hypergraph is currently receiving, while the *extended message set function* returns all messages an object of a specific type may receive. Notice that message redefinition within objects, expressed by the relation $redef^*$ on the type hierarchy, must be taken into account, since the redefinition of a class method implies that only the redefined method can be seen within the scope of a specialized class.

For any hierarchical graph $\langle G, t, \mathcal{H}_\mathcal{M} \rangle$ there is a total function $t^*_E : 2^{E_G} \rightarrow 2^{E_\mathcal{M}}$, which can be viewed as the extension of the typing function to edge (or node) sets. The function t^*_E, when applied to a set $E \in 2^{E_G}$, returns the set $\{t_E(e) \in E_\mathcal{M} | e \in E\} \in 2^{E_\mathcal{M}}$. Notation $t^*_E|_\text{msg}$ and $t^*_E|_\text{attr}$ will be used to denote the application of t^*_E to sets containing exclusively message and attribute (respectively) hyperarcs. Now, given the functions already defined, we can present a definition of the kind of graph which represents an object-oriented system.

Definition 9 (Object-oriented graph). *Let $\mathcal{H}_\mathcal{M}$ be a type hierarchy. A hierarchical hypergraph $\langle G, t, \mathcal{H}_\mathcal{M} \rangle$ is an* object-oriented graph *if and only if all squares in the diagram (in **Set**)*

$$\begin{tikzcd}
2^{E_G} \arrow[d, mapsto, "t^*_E|_\text{msg}"'] & V_G \arrow[l, mapsto, "msg_G"'] \arrow[r, mapsto, "attr_G"] \arrow[d, mapsto, "t_V"'] & 2^{E_G} \arrow[d, mapsto, "t^*_E|_\text{attr}"] \\
2^{E_\mathcal{M}} & V_\mathcal{M} \arrow[l, mapsto, "msg^*_\mathcal{M}"'] \arrow[r, mapsto, "attr^*_\mathcal{M}"] & 2^{E_\mathcal{M}}
\end{tikzcd}$$

commute. If, for each $v \in V_G$, the function $t^*_E|_\text{attr}(attr_G(v))$ is injective, $G^\mathcal{H}$ is said a strict *object-oriented graph. If $t^*_E|_\text{attr}(attr_G(v))$ is also surjective, $G^\mathcal{H}$ is called a* complete *object-oriented graph.*

It is important to realize what sort of message is allowed to target a vertex on an object-oriented graph. The left square on the diagram presented in Definition 9 ensures that an object can only have a message edge targeting it if that message is typed over one of those returned by the extended message set function. It means that the only messages allowed are the least ones in the redefinition chain to which the typing message belongs. This is compatible with the notion of *dynamic binding*, since the method actually called by any object is determined by the actual object present at a certain point of the computation.

Object-oriented graphs can also be *strict* or *complete*. Strict graphs require that nodes do not possess two arcs typed as the same element on the underlying object-model graph. The requirement concerned the injectivity of t_E^* guarantees that there will be no such exceeding attribute connected to any vertex. For an object-oriented graph to be *complete*, however, it is also necessary that all attributes defined on all levels along the type hierarchy (via relation isa^*) are present. The definition of a complete object-oriented graph is coherent with the notion of inheritance within object-oriented framework, since an object inherits all attributes, and exactly those, from its primitive classes.

Object-oriented systems are often composed by a large number of objects, which can receive messages from other objects (including themselves) and react to the messages received. Object-oriented graphs also may have as many objects as desired, since the number and type of attributes (arcs) in each object (vertex) is limited, but the number and type of vertices in the graph representing the system is restricted only by the typing morphism.

Object-oriented graphs are just a special kind of hierarchical hypergraphs. It can be proved the existence of a subcategory of $\mathbf{GraphP}(\mathcal{H}_\mathcal{M})$, $\mathbf{OOGraphP}$ $(\mathcal{H}_\mathcal{M})$, which has object-oriented graphs as objects and hierarchical graph morphisms as arrows.

3 Object-Oriented Graph Grammars

Complete object-oriented graphs (Definition 9) can model an object-oriented system. However, in order to capture the system evolution through time, we need a graph grammar formalism to be introduced.

A *graph production*, or simply a *rule*, specifies how a system configuration may change. A rule has a *left-hand side* and a *right-hand side*, which are both strict object-oriented graphs, and a hierarchical graph morphism to determine what should be altered. Intuitively, a system configuration change occurs in the following way: all items belonging to the left-hand side must be present at the current state to allow the rule to be applied; all items mapped from the left to the right-hand side (via the graph morphism) will be preserved; all items not mapped will be deleted from the current state; and all items present in the right-hand side but not in the left-hand side will be added to the current state to obtain the next one.

Rule restrictions may vary, depending on what is intended for them to represent/implement. Unrestricted rules give rise to a very powerful system in terms of representation capabilities, but they also lead to many undecidable problems. Restrictions are needed not just to make interesting problems decidable (which is important *per se*) but also to reflect restrictions presented in the real system we are modeling. All rule restrictions presented in this text are object-oriented programming principles, as described next.

First of all, no object may have its type altered nor can any two different elements be identified by the rule morphism. This is accomplished by requiring the rule morphism to be injective on nodes and arcs (different elements cannot be merged by the rule application), and the mapping on nodes to be invertible (object types are not modified).

The left-hand side of a rule is required to contain exactly one element of type message, and this particular message must be deleted by the rule application, i.e., each rule represents an object reaction to a message which is consumed in the process. This demand poses no restriction, since systems may have many rules specifying reactions to the same type of message (non-determinism) and many rules can be applied in parallel if their triggers are present at an actual state and the referred rules are not in conflict [4]. Systems' concurrent capabilities are so expressed by the grammar rules, which can be applied concurrently (accordingly to the graph grammar semantics), so one object can treat any number of messages at the same time.

Additionally, only one object having attributes will be allowed on the left-hand side of a rule, along with the requirement that this same object must be the target of the above cited message. This restriction implements the principle of *information hiding*, which states that the internal configuration (implementation) of an object can only be visible, and therefore accessed, by itself.

Finally, although message attributes can be deleted (so they can have their value altered[4]), a corresponding attribute must be added to the rule's right-hand side, in order to prevent an object from gaining or losing attributes along the computation. Notice that this is a *rule* restriction, for if a vertex is deleted, its incident edges will also be deleted. This situation will be explored next, as different kinds of rules are defined.

Definition 10 (Basic object-oriented rule). *A* basic object-oriented rule *is a tuple $\langle L^{\mathcal{H}}, r, R^{\mathcal{H}}\rangle$ where $L^{\mathcal{H}} = \langle L, t_L, \mathcal{H}_{\mathcal{M}}\rangle$ and $R^{\mathcal{H}} = \langle R, t_R, \mathcal{H}_{\mathcal{M}}\rangle$ are strict object-oriented graphs and $r = \langle r_V, r_E\rangle : L^{\mathcal{H}} \to R^{\mathcal{H}}$ is a hierarchical graph morphism holding the following properties:*

- *r_V is injective and invertible, r_E is injective,*
- *$\{v \in V_L | e \in E_L, src_L(e) = v, lab_{\mathcal{M}}(t_L(e)) = \text{attr}\}$ is a singleton, whose unique element is called* attribute vertex*,*
- *$\{e \in E_L | lab_{\mathcal{M}}(t_L(e)) = \text{msg}\}$ is a singleton, whose unique element is called* left-side message*, having as target object the attribute vertex,*
- *the left-side message does not belong to the domain of r, and*
- *for all $v \in V_L$ there is a bijection $b : \{e \in E_L | src_L(e) = v, lab_{\mathcal{M}}(t_L(e)) = \text{attr}\} \leftrightarrow \{e \in E_R | src_R(e) = r_V(v), lab_{\mathcal{M}}(t_R(e)) = \text{attr}\}$, such that $t_R \circ b = t_L$ and $t_L \circ b^{-1} = t_R$.*

[4] Graphs can be enriched with algebras in order to deal with sorts, values and operations. Although we do not develop these concepts here, they can easily be added to this framework.

Different kinds of rules can be defined based on basic object-oriented rules. We define three of them: strict object-oriented rules (Definition 11) do not allow for object creation of deletion; object-oriented rules with creation (Definition 12) allow the creation of new objects; and general object-oriented rules (Definition 13) permit both creation and deletion operations.

Definition 11 (Strict object-oriented rule). *A* strict object-oriented rule *is a basic object-oriented rule $\langle L^{\mathcal{H}}, r, R^{\mathcal{H}} \rangle$ where the hierarchical graph morphism $r = \langle r_V, r_E \rangle$ is such that r_V is total and surjective.*

A strict object-oriented rule presents only the restrictions connected to the object-oriented programming paradigm, along with restrictions to assure that no object is ever created or deleted along the computation. This goal is achieved by requiring a bijection between the vertex sets.

Definition 12 (Object-oriented rule with object creation). *An* object-oriented rule with object creation *is a basic object-oriented rule $\langle L^{\mathcal{H}}, r, R^{\mathcal{H}} \rangle$ where r_V is total, and for all $v \in V_R$, if $v \notin im(r_V)$ the diagram*

$$\begin{tikzcd}
2^{E_R} \arrow[d, "t_E^*|_{\text{msg}}"'] & V_R \arrow[l, "msg_R"'] \arrow[r, "attr_R"] \arrow[d, "t_V"'] & 2^{E_R} \arrow[d, leftrightarrow, "t_E^*|_{\text{attr}}"] \\
2^{E_{\mathcal{M}}} & V_{\mathcal{M}} \arrow[l, "msg_{\mathcal{M}}^*"'] \arrow[r, "attr_{\mathcal{M}}^*"] & 2^{E_{\mathcal{M}}}
\end{tikzcd}$$

commutes and t_E^* is a bijection.

Object-oriented rules with object creation differ from strict object-oriented rules in two aspects: r_V is not necessarily surjective, so new vertices can be added by the rule, and all created vertices must have exactly the attributes defined along its type hierarchy.

Definition 13 (General object-oriented rule). *A general object-oriented rule is a object-oriented rule with object creation $\langle L^{\mathcal{H}}, r, R^{\mathcal{H}} \rangle$ where $dom(r_V) = V_L$ or $dom(r_V) = V_L \backslash \{\text{attribute vertex}\}$.*

General object-oriented rules allow object deletion. Notice, however, that an object can only delete itself.

Different types of rules give rise to different possible computations. The more restrictive the rules, the more easier becomes to derive properties from system computations. Verification of computational power of rules is, however, beyond the scope of this paper.

Definition 14 (Object-oriented match). *Given a strict object-oriented graph $G^{\mathcal{H}}$ and an object-oriented rule $\langle L^{\mathcal{H}}, r, R^{\mathcal{H}} \rangle$, an* object-oriented match *between $L^{\mathcal{H}}$ and $G^{\mathcal{H}}$ is a hierarchical graph morphism $m = \langle m_V, m_E \rangle : L^{\mathcal{H}} \to G^{\mathcal{H}}$ such that m_V is total, m_E is total and injective, and for any two elements $a, b \in L$, if $m(a) = m(b)$ then either $a, b \in dom(r)$ or $a, b \notin dom(r)$.*

The role of a *match* is to detect a situation when a rule can be applied. It occurs whenever a rule's left-hand side is present somewhere within the system graph. Notice that distinct vertices can be identified by the matching morphism. This is sensible, since an object can point to itself through one of its attributes, or pass itself as a message parameter to another object. However, it would make no sense to identify different attributes or messages, so the edge component of the matching morphism is required to be injective. Additionally, preserved elements cannot be identified with deleted elements.

The purpose of method redefinition is to take advantage of the polymorphism concept through the mechanism known as *dynamic binding.* Dynamic binding is usually implemented in object-oriented languages by a function pointer virtual table, from which it is decided which method should be called at that execution point. This decision is modelled in our work by a retyping message function.

Definition 15 (Retyping function ret**).** *Let* $\mathcal{H}_{\mathcal{M}} = \langle \mathcal{M}, isa, redef \rangle$ *be a type hierarchy and let* $G^{\mathcal{H}} = \langle G, t, \mathcal{H}_{\mathcal{M}} \rangle$ *be a hierarchical graph. The retyping function, when applied over a message hyperedge* $m \in E_G$*, i.e.,* $(lab_{\mathcal{M}} \circ t_E)(m) = \text{msg}$*, and over a node* $v \in V_G$ *such that* $(t_V(v), (t_V \circ tar_G)(m)) \in isa^*$*, returns a hyperedge* m' *where* $src_G(m') = src_G(m)$*,* $tar_G(m') = v$ *e* $t_E(m') = e \in (msg^*_{\mathcal{M}} \circ t_V)(v)$*, such that* $(e, t_E(m)) \in redef^*$*.*

It can be shown that the retyping function is well defined (i.e., the set $(msg^*_{\mathcal{M}} \circ t_V)(v)$ is always a singleton). It can also be shown that the message passed to the function is retyped as the least type within the redefinition chain (with respect relation $redef^*$) to which that message belongs.

A *derivation step*, or simply a *derivation*, will represent a discrete system change in time, i.e., a rule application over an actual system specified as a graph.

Definition 16 (Object-oriented derivation). *Given a strict object-oriented graph* $G^{\mathcal{H}} = \langle G, t_G, \mathcal{H}_{\mathcal{M}} \rangle$*, an object-oriented rule* $\langle \langle L, t_L, \mathcal{H}_{\mathcal{M}} \rangle, r, \langle R, t_R, \mathcal{H}_{\mathcal{M}} \rangle \rangle$*, and an object-oriented match* $m : L^{\mathcal{H}} \to G^{\mathcal{H}}$*, their* object-oriented derivation *can be computed in two steps:*

1. *Construct the pushout of* $r : L \to R$ *and* $m : L \to G$ *in* ***GraphP****,* $\langle H, r' : G \to H, m' : R \to H \rangle$ *[4];*
2. *construct the strict object-oriented graph* $H^{\mathcal{H}} = \langle H, t_H, \mathcal{H}_{\mathcal{M}} \rangle$ *where, for each* $v \in V_H$*,* $t_H(v) = glb_{isa^*}({r'}^{-1}(v) \cup {m'}^{-1}(v))$*, for each* $e \in E_H|_{\text{attr}}$*,* $t_H(e) = glb_{redef^*}({r'}^{-1}(e) \cup {m'}^{-1}(e))$*, for each* $e \in E_H|_{\text{msg}}$*,* $t_H(e) = t_G(e)$ *if* $r'(x) = e$ *for some* $x \in E_G$*, or* $t_H(e) = ret({m'}^{-1}(e), tar_H(e))$ *if* $m'(x) = e$ *for some* $x \in E_R$*.*

The tuple $\langle H^{\mathcal{H}}, r', m' \rangle$ *is then the resulting derivation of rule* r *at match* m*.*

An object-oriented derivation collapses the elements identified simultaneously by the rule and by the match. Element typing, needed to transform the resulting hypergraph into an object-oriented graph is done by getting the greatest lower bound (with respect the partial order relations isa^* and $redef^*$) of the elements mapped by morphisms m' and r' to the same element. The basic object-oriented rule restriction concerning object types (which cannot be altered by the rule) assures that it always exist. Messages, however, need some extra care. Since graph L presents a single message, which is deleted by the rule application, a message on H comes from either G or R. If it comes from G, which is an object-oriented graph itself, no retyping is needed. However, if it comes from R, in order to assure that H is also an object-oriented graph, it must be retyped according to the type of the element it is targeting on the graph H.

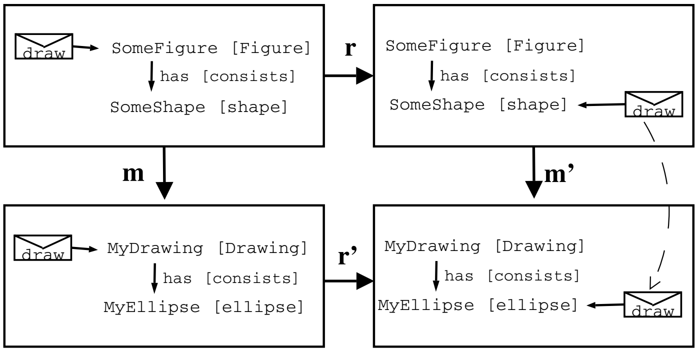


Fig. 3. Dynamic binding as message retyping

Figure 3 shows a situation where the need for a message retyping is made clear. As usual, the typing morphism is shown between brackets. Rule r portrays a common situation: the action resulting from a method calling is the calling of another method from one of the object's attributes. Here, a *figure* is drawn by making its constituent *shape* be drawn. However, since the rule's left-hand side is matched to a drawing which has an *ellipse* as constituent, and since the method *Draw* is redefined within that level, the resulting message cannot be typed as a *shape Draw*, but as an *ellipse Draw* (indicated by the only explicit arrow from R to H). Notice that m' is still a hierarchical graph morphism (although it is not strict). Hence, message retyping is the construction that implements dynamic binding on the computational process.

It can be shown that the an object-oriented derivation is a pushout structure in the category $\mathbf{OOGraphP}(\mathcal{H}_{\mathcal{M}})$.

Given the graph structures presented earlier and the rules to be applied to them, some interesting properties can be demonstrated. Closure properties are especially interesting, such as the ones expressed by the theorems below.

Theorem 1. *The class of complete object-oriented graphs is closed under object-oriented derivation using strict object-oriented rules.*

Proof. (sketch) Let $G^{\mathcal{H}}$ be a complete object-oriented graph, $\langle L^{\mathcal{H}}, r, R^{\mathcal{H}}\rangle$ be a strict object-oriented rule and $\langle L^{\mathcal{H}}, m, G^{\mathcal{H}}\rangle$ be an object-oriented match, and

$\langle L^{\mathcal{H}}, r', m' \rangle$ the resulting derivation of rule r at match m. Being r_V is a total bijection, for any $v \in V_L$ $t_L(v) = t_R(r_V(v))$ holds. Since m is a hierarchical graph morphism, for any vertex $v \in dom(r_V) \cap dom(m_V)$, $(t_G \circ m_V(v), t_R \circ r_V(v)) isa^*$, and so $t_H \circ r'_V \circ m_V(v) = t_H \circ m'_V \circ r_V(v) = t_G \circ m_V(v)$. So, V_H is isomorphic to V_G.

Now, let b be the bijection existing between the attribute edges from $A_L \subseteq E_L$ and $A_R \subseteq E_R$ defined as the last basic object-oriented rule restriction in Definition 10. Notice b is defined over *all* attribute edges of both graphs L and R. The match m between the rule's left-side and graph G is total on vertices and arcs, and injective on arcs, and by the characteristics of the pushout construction, function m' is also total and injective on arcs. Notice that all edges from G are either belonging to the image of m_E (the mapped edges) or not (the context edges). Since the context edges are mapped unchanged to the graph H (and so there is a natural bijection between them), it must exist a bijection $B : E_G \leftrightarrow E_H$ which implies the existence of the trivial bijection $2^B : 2^{E_G} \rightarrow 2^{E_H}$, and since the sets V_G and V_H are equal (up to isomorphism), it can be concluded that $H^{\mathcal{H}}$ is a complete object-oriented graph. □

Theorem 2. *The class of complete object-oriented graphs is closed under object-oriented derivation using object-oriented rules with object creation.*

Proof. (sketch) The same reasoning applied to the proof of Theorem 1 can be used to show that, in this case, V_G is isomorphic to a subset of V_H. The additional vertices of V_H are those created by the rule application (i.e., those isomorphic to the set $\{v \in V_R \mid v \notin im(r_V)\}$). But since all $v \notin im(r_V)$ is required to behave like a complete object-oriented graph when considered alone, so its inclusion on H will assure, along with Theorem 1, that it is also a complete object-oriented graph. □

Theorem 3. *The class of complete object-oriented graphs is not closed under object-oriented derivation using general object-oriented rules.*

Proof. (sketch) This theorem can be easily proven by a counterexample, since the deletion of a node causes the deletion on any of its incident arcs. The resulting graph will not be a complete object-oriented graph anymore. □

Theorem 3 describes a situation known as deletion in unknown contexts. This situation is very common in distributed systems, where the deletion of an object causes a number of dangling pointers to occur in the system as a whole. So, rules that allow object deletion can be used to find this kind of undesirable situations within a specification.

An interesting side effect derived from the use of rules that allow object deletion is that any dangling pointer would cause a edge cease to exist. In this case, any rule which takes that particular edge into consideration can no longer be applied (for no match can be found for that rule). When modeling system execution, this situation leads to the prevention of an execution runtime error, which would occur if an attempt to access an object which is no longer there is made.

Definition 17 (Object-oriented graph grammar). *An object-oriented graph grammar is a tuple $\langle I^{\mathcal{H}}, P^{\mathcal{H}}, \mathcal{H}_{\mathcal{M}} \rangle$ where $I^{\mathcal{H}}$ is a complete object-oriented graph, $P^{\mathcal{H}}$ is a finite set of object-oriented rules, and $\mathcal{H}_{\mathcal{M}}$ is a type hierarchy.*

Graph $I^{\mathcal{H}}$ portrays the initial system configuration. The system can evolve through the application of the productions in the grammar. All possible system future configurations are given by the set $\{G^{\mathcal{H}} \mid I^{\mathcal{H}} \Rightarrow^* G^{\mathcal{H}}\}$.

4 Conclusions

This work presented a first step towards a very high level and intuitive formal framework compatible with the main principles of object-oriented specification and programming. More specifically it presents, in terms of object-oriented graphs and morphisms, a way of defining classes, which can be primitive or specialized from others (though the *isa* relationship) together with a graph transformation-based model of computation compatible with polymorphism and dynamic binding, which are fundamental in the object-oriented programming model of execution.

A significant advantage to the use of a formal framework for object-oriented system specification is in the ability to apply rigorous inference rules so as to allow reasoning with the specifications and deriving conclusions on their properties. Fixing the sort of rules to be used within a graph grammar, properties regarding the computational model can be derived. Being this a formal framework, the semantics of operations (such as system and grammar composition) can also be derived.

Graph grammars are well suited for system specification, and object-oriented graph grammars, as presented in this text, fill the need for the key features of object-oriented systems be incorporated into a formal framework.

References

1. Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jorg Kreowski, Sabine Kuske, Detlef Plump, Andy Schurr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34:1–54, 1999.
2. W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *POPL'90 - 17th Annual ACM Symposium on Principles of Programming Languages*. Kluwer Academic Publishers, 1990.
3. Fernando Luís Dotti and Leila Ribeiro. Specification of mobile code using graph grammars. In *Formal Methods for Open Object-Based Distributed Systems IV*. Kluwer Academic Publishers, 2000.
4. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation. part ii: single pushout approach and comparison with double pushout approach. In G. Rozemberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1 – Foundations, chapter 4, pages 247–312. World Scientific, Singapore, 1996.

5. 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, 1973.
6. J. L. Fiadeiro, C. Sernadas, T. Maibaum, and G. Saake. Proof-theoretic semantics of object-oriented specification constructs. In W. Kent, R. Meersman, and S. Khosla, editors, *Object-Oriented Databases: Analysis, Design and Construction*, pages 243–284. North-Holland, 1991.
7. Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1992.
8. Aline Brum Loreto, Leila Ribeiro, and Laira Vieira Toscani. Decodability and tractability of a problem in object-based graph grammars. In *17th IFIP World Computer Congress - Theoretical Computer Science*, Montreal, 2002. Kluwer.
9. Michael Löwe. *Extended Algebraic Graph Transformation*. PhD thesis, Technischen Universität Berlin, Berlin, Feb 1991.
10. Ugo Montanari, Marco Pistore, and Francesca Rossi. Modeling concurrent, mobile and coordinated systems via graph transformations. In H. Ehrig, H-J. Kreowski, U. Montanari, and G. Rozemberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 3 – Concurrency, Parallelism and Distribution, chapter 4. World Scientific, 2000.
11. George A. Papadopoulos. Concurrent object-oriented programming using term graph rewriting techniques. *Information and Software Technology*, (38):539–547, 1996.
12. Leila Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars*. Phd thesis, Technische Universität Berlin, Berlin, June 1996. 202p.
13. David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 3(4), 1991.

A Rewriting Based Model for Probabilistic Distributed Object Systems

Nirman Kumar, Koushik Sen, José Meseguer, and Gul Agha

Department of Computer Science
University of Illinois at Urbana-Champaign
{nkumar5,ksen,meseguer,agha}@cs.uiuc.edu

Abstract. Concurrent and distributed systems have traditionally been modelled using nondeterministic transitions over configurations. The nondeterminism provides an abstraction over scheduling, network delays, failures and randomization. However a probabilistic model can capture these sources of nondeterminism more precisely and enable statistical analysis, simulations and reasoning. We have developed a general semantic framework for probabilistic systems using probabilistic rewriting. Our framework also allows nondeterminism in the system. In this paper, we briefly describe the framework and its application to concurrent object based systems such as actors. We also identify a sufficiently expressive fragment of the general framework and describe its implementation. The concepts are illustrated by a simple client-server example.

Keywords: Rewrite theory, probability, actors, Maude, nondeterminism.

1 Introduction

A number of factors, such as processor scheduling and network delays, failures, and explicit randomization, generally result in nondeterministic execution in concurrent and distributed systems. A well known consequence of such nondeterminism is an exponential number of possible interactions which in turn makes it difficult to reason rigorously about concurrent systems. For example, it is infeasible to use techniques such as model checking to verify any large-scale distributed systems. In fact, some distributed systems may not even have a finite state model: in particular, *networked embedded systems* involving *continuously* changing parameters such as time, temperature or available battery power are infinite state.

We believe that a large class of concurrent systems may become amenable to a rigorous analysis if we are able to quantify some of the probabilities of transitions. For example, network delays can be represented by variables from a probabilistic distribution that depends on some function of the system state. Similarly, available battery power, failure rates, etc., may also have a probabilistic behavior. A probabilistic model can capture the statistical regularities in such systems and enable us to make probabilistic guarantees about its behavior.

We have developed a model based on rewriting logic [11] where the rewrite rules are enriched with probability information. Note that rewriting logic provides a natural model for object-based systems [12]. The local computation of each object is modelled by rewrite rules for that object and one can reason about the global properties that result from the interaction between objects: such interactions may be asynchronous as in actors, or synchronous as in the π-calculus. In [9] we show how several well known models of probabilistic and nondeterministic systems can be expressed as special cases of probabilistic rewrite theories. We also propose a temporal logic to express properties of interest in probabilistic systems. In this paper we show how probabilistic object systems can be modelled in our framework. Our *probabilistic rewriting* model is illustrated using a client-server example. The example also shows how nondeterminism, for which we do not have the probability distributions, is represented naturally in our model. Nondeterminism is eventually removed by the system *adversary* and converted into probabilities in order to define a probability space over computation paths.

The *Actor* model of computation [1] is widely used to model and reason about object-based distributed systems. Actors have previously been modelled as rewrite theories [12]. Probabilistic rewrite theories can be used to model and reason about actor systems where actors may fail and messages may be dropped or delayed and the associated probability distributions are known (see Section 3).

The rest of this paper is organized as follows. Section 2 provides some background material on membership equational logic [13] and rewriting [11] as well as probability theory. Section 3 starts by giving an intuitive understanding of how a step of computation occurs in a probabilistic rewrite theory. We then introduce an example to motivate the modelling power of our framework and formalize the various concepts. In Section 4 we define an important subclass of probabilistic rewrite theories, and in Section 5, we describe its Maude implementation. The final section discusses some directions for future research.

2 Background and Notation

A *membership equational theory* [13] is a pair (Σ, E), with Σ a *signature* consisting of a set K of *kinds*, for each $k \in K$ a set S_k of *sorts*, a set of *operator* declarations of the form $f : k_1 \dots k_n \to k$, with $k, k_1, \dots, k_n \in K$ and with E a set of *conditional Σ-equations* and *Σ-memberships* of the form

$$(\forall \overrightarrow{x})\; t = t' \Leftarrow u_1 = v_1 \wedge \dots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \dots \wedge w_m : s_m$$
$$(\forall \overrightarrow{x})\; t : s \Leftarrow u_1 = v_1 \wedge \dots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \dots \wedge w_m : s_m$$

The $\overrightarrow{x}$ denote *variables* in the terms t, t', u_i, v_i and w_j above. A membership $w : s$ with w a Σ-term of kind k and $s \in S_k$ asserts that w has sort s. Terms that do not have a sort are considered *error* terms. This allows membership equational theories to specify partial functions within a total framework. A *Σ-algebra* B consists of a K-indexed family of sets $X = \{B_k\}_{k \in K}$, together with

1. for each $f : k_1 \dots k_n \to k$ in Σ a function $f_B : B_{k_1} \times \dots \times B_{k_n} \to B_k$
2. for each $k \in K$ and each $s \in S_k$ a subset $B_s \subseteq B_k$.

We denote the algebra of terms of a membership equational theory by T_Σ. The *models* of a membership equational theory (Σ, E) are those Σ-algebras that satisfy the equations E. The inference rules of membership equational logic are *sound* and *complete* [13]. Any membership equational theory (Σ, E) has an *initial algebra* of terms denoted $T_{\Sigma/E}$ which, using the inference rules of membership equational logic and assuming Σ *unambiguous* [13], is defined as a quotient of the term algebra T_Σ by

- $t \equiv_E t' \quad \Leftrightarrow \quad E \vdash (\forall\emptyset)\ t = t'$
- $[t]_{\equiv_E} \in T_{\Sigma/E,s} \quad \Leftrightarrow \quad E \vdash (\forall\emptyset)\ t : s$

In [2] the usual results about *equational simplification*, *confluence*, *termination*, and *sort-decreasingness* are extended in a natural way to membership equational theories . Under those assumptions a membership equational theory can be executed by equational simplification using the equations from left to right, perhaps modulo some *structural* (e.g. associativity, commutativity and identity) axioms A. We denote the algebra of terms simplified by equations and structural axioms as $T_{\Sigma,E\cup A}$ and the isomorphic algebra of equivalence classes modulo axioms A, of equationally simplified terms by $Can_{\Sigma,E/A}$. The notation $[t]_A$ represents the equivalence class of a term t fully simplified by the equations.

In a standard *rewrite theory* [11], transitions in a system are described by labelled rewrite rules of the form

$$l : t(\overrightarrow{x}) \longrightarrow t'(\overrightarrow{x})\ \mathit{if}\ C(\overrightarrow{x})$$

Intuitively, a rule of this form specifies a *pattern* $t(\overrightarrow{x})$ such that if some fragment of the system's state matches that pattern and satisfies the condition C, then a local transition of that state fragment, changing into the pattern $t'(\overrightarrow{x})$ can take place. In a *probabilistic rewrite rule* we add probability information to such rules. Specifically, our proposed probabilistic rules are of the form,

$$l : t(\overrightarrow{x}) \longrightarrow t'(\overrightarrow{x}, \overrightarrow{y})\ \mathit{if}\ C(\overrightarrow{x})\ \mathit{with\ probability}\ \pi(\overrightarrow{x}).$$

In the above, the set of variables in the left-hand side term $t(\overrightarrow{x})$ is $\overrightarrow{x}$, while some new variables $\overrightarrow{y}$ may be present in the term $t'(\overrightarrow{x}, \overrightarrow{y})$ on the right-hand side. Of course it is not necessary that *all* of the variables in $\overrightarrow{x}$ occur in $t'(\overrightarrow{x}, \overrightarrow{y})$. The rule will match a state fragment if there is a substitution θ for the variables $\overrightarrow{x}$ that makes $\theta(t)$ equal to that state fragment and the condition $\theta(C)$ is true. Because the right-hand side $t'(\overrightarrow{x}, \overrightarrow{y})$ may have new variables $\overrightarrow{y}$, the next state is not *uniquely* determined: it depends on the choice of an additional substitution ρ for the variables $\overrightarrow{y}$. The choice of ρ is made according to the probability function $\pi(\theta)$, where π is not a fixed probability function, but a *family* of functions: one for each match θ of the variables $\overrightarrow{x}$.

The Maude system [4, 5] provides an execution environment for membership equational and rewrite theories. The Full Maude [6] library built on top of the Core Maude environment allows users to specify object oriented modules in a convenient syntax. Several examples in [12, 5] show specifications of object based systems in Maude. The code for our example in Section 3 is written in the syntax of Maude 2.0 [5].

To succinctly define probabilistic rewrite theories, we use a few basic notions from axiomatic probability theory. A σ-algebra on a set X is a collection $\mathcal{F}$ of subsets of X, containing X itself and closed under complementation and finite or countably infinite unions. For example the power set $\mathcal{P}(X)$ of a set X is a σ-algebra on X. The elements of a σ-algebra are called *events*. We denote by $\mathcal{B}_{\mathbb{R}}$ the smallest σ-algebra on $\mathbb{R}$ containing the sets $(-\infty, x]$ for all $x \in \mathbb{R}$. We also remind the reader that a *probability space* is a triple $(X, \mathcal{F}, \pi)$ with $\mathcal{F}$ a σ-algebra on X and π a *probability measure function*, defined on the σ-algebra $\mathcal{F}$ which evaluates to 1 on X and distributes by addition over finite or countably infinite union of disjoint events. For a given σ-algebra $\mathcal{F}$ on X, we denote by $PFun(X, \mathcal{F})$ the set

$$\{\pi \mid (X, \mathcal{F}, \pi) \text{ is a probability space}\}$$

Definition 1 ($\mathcal{F}$-Cover). *For a σ-algebra $\mathcal{F}$ on X, an $\mathcal{F}$-cover is a function $\alpha : X \to \mathcal{F}$, such that $\forall x \in X \;\; x \in \alpha(x)$.*

Let π be a probability measure function defined on a σ-algebra $\mathcal{F}$ on X, and suppose α is an $\mathcal{F}$-cover. Then notice that $\pi \circ \alpha$ naturally defines a function from X to $[0, 1]$. Thus, for example, for $X = \mathbb{R}$ and $\mathcal{F} = \mathcal{B}_{\mathbb{R}}$, we can define α to be the function that maps the real number x to the set $(-\infty, x]$. With X a finite set and $\mathcal{F} = \mathcal{P}(X)$, the power set of X, it is natural to define α to be the function that maps $x \in X$ to the singleton $\{x\}$.

3 Probabilistic Rewrite Theories

A probabilistic rewrite theory has an interleaving execution semantics. A step of computation changes a term $[u]_A$ to $[v]_A$ by the application of a single rewrite rule on some subterm of the given *canonical* term $[u]_A$. Recall the form of a probabilistic rewrite rule as described in the previous section. Firstly, all context, rule, substitution (for the variables $\overrightarrow{x}$) triples arising from possible applications of rewrite rules (see definition 6) to $[u]_A$ are computed. One of them $([\mathbb{C}]_A, r, [\theta]_A)$ (for the justification of the A subscript see definitions 2, 3 and 5) is chosen *nondeterministically*. This step essentially represents the nondeterminism in the system. After that has been done, a particular substitution $[\rho]_A$ is chosen *probabilistically* for the new variables $\overrightarrow{y}$ and $[\rho]_A$ along with $[\theta]_A$, is applied to the term $t'(\overrightarrow{x}, \overrightarrow{y})$ and placed inside the context $\mathbb{C}$ to obtain the term $[v]_A$. The choice of the new substitution $[\rho]_A$ is from the set of possible substitutions for $\overrightarrow{y}$. The probabilities are defined as a *function* of $[\theta]_A$. This gives the framework great expressive power. Our framework can model both nondeterminism and probability in the system. Next we describe our example, model it as an object based rewrite theory and indicate how the rewrite rules model the probabilities and nondeterminism.

A Client-Server Example. Our example is a situation where a client is sending computational jobs to servers across a network. There are two servers S_1 and

```
pmod QOS-MODEL is
      ...
      vars $L$ $N$ $m_1$ $m_2$: Nat.
      vars Cl Sr Nw: Oid.
      var i: Bit.
      vars C Q: Configuration.
      var M: Msg.
      op _ ← _: Oid Nat → Msg.
      class Client   |sent:Nat, svc$_1$:Nat, svc$_2$:Nat.
      class Network  |soup:Configuration.
      class Server   |queue:Configuration.
      ops H Nt $S_1$ $S_2$: → Oid.
      ops acq$_1$ acq$_2$: → Msg.
      prl [req]:⟨Cl:Client|sent:$N$, svc$_1$:$m_1$, svc$_2$:$m_2$⟩⟨Nw: Network|soup:C⟩⇒
       ⟨Cl: Client|sent:$(N+1)$, svc$_1$:$m_1$, svc$_2$:$m_2$⟩⟨Nw: Network|soup:C (Sr ← $L$)⟩.
      cprl [acq]:⟨Cl:Client|svc$_1$:$m_1$, svc$_2$:$m_2$⟩⟨Nw:Network|soup:M C⟩⇒
       ⟨Cl:client|svc$_1$:$m_1 + \delta(i, M, 1)$, svc$_2$:$m_2 + \delta(i, M, 2)$⟩⟨Nw:Network|soup:C⟩
        if acq(M).
      prl [deliver]:⟨Nw:Network|soup:(Sr ← $L$) C⟩⟨Sr:Server|queue:Q⟩⇒
       ⟨Nw:Network|soup:C⟩⟨Sr:Server|queue:Q M⟩.
      prl [process]:⟨Sr:Server|queue:(Sr ← $L$) Q⟩⟨Nw:Network|soup:C⟩ ⇒
       ⟨Sr:Server|queue:Q⟩⟨Nw:Network|soup:C M⟩.
endpm
```

Fig. 1. A client-server example.

S_2. S_1 is computationally more powerful than S_2, but the network connectivity to S_2 is better (more reliable) than that to S_1 and packets to S_1 may be dropped without being delivered, more frequently than packets to S_2. The servers may also drop requests if the load increases beyond a certain threshold. The computationally more powerful server S_1 drops packets with a lower probability than S_2. We would like to reason about a good randomized policy for the client. The question here is: which server is it better to send packets to, so that a larger fraction of packets are processed rather than dropped? Four objects model the system. One of them, the client, sends packets to the two server objects deciding probabilistically before each send which server to send the packet to. The other object models a network, which can either transmit the packets correctly, drop them or deliver them out of order. The remaining two objects are server objects which either drop a request or process it and send an acknowledgement message. The relevant fragment of code specifying the example is given in Figure 1. The client object named H maintains the total number of requests sent in a variable *sent* and those which were successfully processed by servers S_1, S_2 in variables svc_1 and svc_2 respectively. Notice that for example in $svc_1 : m_1$ the m_1 is the *value* of the variable named svc_1. An example term representing a possible system state is

⟨ H : Client | sent : 3, svc$_1$: 1, svc$_2$: 0 ⟩ ⟨ Nt : Network | soup : (S_1 ← 10) ⟩
⟨ S_1 : Server | queue : nil ⟩ ⟨ S_2 : Server | queue : (S_2 ← 5) ⟩

The term above of sort *Configuration* (collection of objects and messages) represents a multiset of objects combined with an empty syntax (juxtaposition) multiset union operator that is declared associative and commutative. The client has sent 3 requests in total, out of which one has already been serviced by S_1, one is yet to be delivered and one request is yet pending at the server S_2. The numbers 10 and 5 represent the measure of the loads in the respective requests.

We discuss the rules labelled *req* and *acq*. Henceforward we refer to a rule by its label. Though not shown in Figure 1, a probabilistic rewrite theory associates some functions with the rules, defining the probabilities. The rule *req* models the client sending a request to one of the servers by putting a message into the network object's variable *soup*. The rule involves two new variables Sr and L on the right-hand side. Sr is the name of the server to which the request is sent and L is the message *load*. A probability function $\pi_{req}(Cl, N, m_1, m_2, Nw, C)$ associated with the rule *req* (see definition 4) will decide the distribution of the new variables Sr and L, and thus the randomized policy of the client. For example, it can assign higher probability values to substitutions with $Sr = S_1$, if it finds that $m_1 > m_2$; this would model a heuristic policy which sends more work to the server which is performing better. In this way the probabilities can depend on the values of m_1, m_2 (and thus the state of the system). In the rule labelled *acq* there is only one new variable i on the right-hand side. That variable can only assume two values $0, 1$ with nonzero probability. 0 means a message drop, so that $\delta(0, M, 1) = \delta(0, M, 2) = 0$, while if $i = 1$ then the appropriate *svc* variable is incremented. The distribution of i as decided by the function $\pi_{acq}(\ldots, M)$ could depend on M, effectively modelling the network connectivity. The network drops messages more frequently for $M = acq_1$ (an acq message from server S_1) than it does for $M = acq_2$. Having the distribution of new variables *depend* on the substitution gives us the ability to model general distributions. The associativity and commutativity attribute of the juxtaposition operator for the sort *Configuration* essentially allows nondeterminism in the order of message delivery by the network (since it chooses a message to process, from the associative commutative *soup* of messages) and the order of messages processed by the servers.

The more frequently the rewrite rules for the network object are applied (which allow it to process the messages in it soup), the more frequently the *acq* messages will be delivered. Likewise, the more frequently the rewrite rules for a particular server are applied, the more quickly will it process its messages. Thus, during a computation the values m_1, m_2, which determine the client's randomized policy, will actually depend not only on the probability that a server correctly processes the packets and the network correctly delivers requests and acknowledgments, but also on how frequently the appropriate rewrite rules are applied. However, the exact frequency of application depends on the *nondeterministic* choices made. We can now see how the nondeterminism effectively influences the probabilities in the system. As explained later, the nondeterminism is removed (converted into probabilities) by what is called an *adversary* of the system. In essence the adversary is like a *scheduler* which determines the rate of progress of each component. The choice of adversary is important for the behavior of the system. For example, we may assume a *fair* adversary that chooses between its nondeterministic choices equally frequently. At an intuitive level this would mean that the different parts of the system compute at the same rate. Thus, it must be understood that the model defined by a probabilistic rewrite theory is parameterized on the adversary. The system modeler must define the adversary based on an understanding of how frequently different objects in the system advance. Model checking of probabilistic systems quantifies over adversaries, whereas a simulation has to fix an adversary.

We now define our framework formally.

Definition 2 (E/A-Canonical Ground Substitution). *An E/A-canonical ground substitution is a substitution $\theta : \overrightarrow{x} \to T_{\Sigma,E\cup A}$.*

Intuitively an E/A-canonical ground substitution represents a substitution of ground terms from the term algebra T_Σ for variables of the corresponding sorts, so that all of the terms have already been reduced as much as possible by the equations E and the structural axioms A. For example the substitution 10×2 to a variable of sort *Nat is not* a canonical ground substitution, but a substitution of 20 for the same variable is a canonical ground substitution.

Definition 3 (A-Equivalent Substitution). *Two E/A-canonical ground substitution $\theta, \rho : \overrightarrow{x} \to T_{\Sigma,E\cup A}$ are A-equivalent if and only if $\forall x \in \overrightarrow{x}\ [\theta(x)]_A = [\rho(x)]_A$.*

We use $CanGSubst_{E/A}(\overrightarrow{x})$ to denote the set of all E/A-canonical ground substitutions for the set of variables $\overrightarrow{x}$. It is easy to see that the relation of A-equivalence as defined above is an equivalence relation on the set $CanGSubst_{E/A}(\overrightarrow{x})$. When the set of variables $\overrightarrow{x}$ is understood, we use $[\theta]_A$ to denote the equivalence class containing $\theta \in CanGSubst_{E/A}(\overrightarrow{x})$.

Definition 4 (Probabilistic Rewrite Theory). *A probabilistic rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E \cup A, R, \pi)$, with $(\Sigma, E \cup A, R)$ a rewrite theory with the rules $r \in R$ of the form*

$$l : t(\overrightarrow{x}) \to t'(\overrightarrow{x}, \overrightarrow{y})\ \textit{if}\ C(\overrightarrow{x})$$

where

- *$\overrightarrow{x}$ is the set of variables in t.*
- *$\overrightarrow{y}$ is the set of variables in t' that are not in t. Thus t' might have variables coming from the set $\overrightarrow{x} \cup \overrightarrow{y}$ but it is not necessary that all variables in $\overrightarrow{x}$ occur in t'.*
- *C is a condition of the form $(\bigwedge_j u_j = v_j) \wedge (\bigwedge_k w_k : s_k)$, that is, C is a conjunction of equations and memberships;*

and π is a function assigning to each rewrite rule $r \in R$ a function

$$\pi_r : [\![C]\!] \to PFun(CanGSubst_{E/A}(\overrightarrow{y}), \mathcal{F}_r)$$

where $[\![C]\!] = \{[\mu]_A \in CanGSubst_{E/A}(\overrightarrow{x}) \mid E \cup A \vdash \mu(C)\}$ *is the set of* E/A*-canonical substitutions for* $\overrightarrow{x}$ *satisfying the condition* C*, and* $\mathcal{F}_r$ *is a* σ*-algebra on* $CanGSubst_{E/A}(\overrightarrow{y})$*. We denote a rule* r *together with its associated function* π_r*, by the notation*

$$l : t(\overrightarrow{x}) \rightarrow t'(\overrightarrow{x}, \overrightarrow{y}) \text{ if } C(\overrightarrow{x}) \text{ with probability } \pi_r(\overrightarrow{x})$$

We denote the class of probabilistic rewrite theories by **PRwTh** . Notice the following points in the definition

1. Rewrite rules may have new variables $\overrightarrow{y}$ on the right-hand side.
2. The condition $C(\overrightarrow{x})$ on the right-hand side depends only on the variables $\overrightarrow{x}$ occurring in the term $t(\overrightarrow{x})$ on the left-hand side.
3. The condition $C(\overrightarrow{x})$ is simply a conjunction of equations and memberships (but no rewrites).
4. $\pi_r(\overrightarrow{x})$ specifies, for each substitution θ for the variables $\overrightarrow{x}$, the probability of choosing a substitution ρ for the $\overrightarrow{y}$. In the next section we explain how this is done.

3.1 Semantics of Probabilistic Rewrite Theories

Let $\mathcal{R} = (\Sigma, E \cup A, R, \pi)$ be a probabilistic rewrite theory such that:

1. E is confluent, terminating and sort-decreasing modulo A [2].
2. the rules R are coherent with E modulo A [4].

We also assume a choice for each rule r of an $\mathcal{F}_r$-cover $\alpha_r : CanGSubst_{E/A}(\overrightarrow{y}) \rightarrow \mathcal{F}_r$. This $\mathcal{F}_r$-cover will be used to assign probabilities to rewrite steps. Its choice will depend on the particular problem under consideration.

Definition 5 (Context). *A* context $\mathbb{C}$ *is a* Σ*-term with a single occurrence of a single variable,* $\odot$*, called the* hole*. Two contexts* $\mathbb{C}$ *and* $\mathbb{C}'$ *are* A*-equivalent if and only if* $A \vdash (\forall \odot)\ \mathbb{C} = \mathbb{C}'$*.*

Notice that the relation of A-equivalence for contexts as defined above, is an equivalence relation on the set of contexts. We use $[\mathbb{C}]_A$ for the equivalence class containing context $\mathbb{C}$. For example the term

$\odot$ ⟨ Nt : Network | soup : ($S_1 \leftarrow 10$) ⟩
⟨ S_1 : Server | queue : nil ⟩ ⟨ S_2 : Server | queue : ($S_2 \leftarrow 5$) ⟩

is a context.

Definition 6 (R/A-Matches). *Given* $[u]_A \in Can_{\Sigma,E/A}$*, its* R/A-matches *are triples* $([\mathbb{C}]_A, r, [\theta]_A)$*, where if* $r \in R$ *is a rule*

$$l : t(\overrightarrow{x}) \rightarrow t'(\overrightarrow{x}, \overrightarrow{y}) \text{ if } C(\overrightarrow{x}) \text{ with probability } \pi_r(\overrightarrow{x})$$

then $[\theta]_A \in [\![C]\!]$*, that is* $[\theta]_A$ *satisfies condition* C *and* $[u]_A = [\mathbb{C}(\odot \leftarrow \theta(t))]_A$*, so* $[u]_A$ *is the same as* θ *applied to the term* $t(\overrightarrow{x})$ *and placed in the context.*

Consider the canonical-term

$\langle$ H : Client | sent : 3, svc$_1$: 1, svc$_2$: 0 $\rangle$ $\langle$ Nt : Network | soup : ($S_1 \leftarrow 10$) $\rangle$
$\langle$ S_1 : Server | queue : nil $\rangle$ $\langle$ S_2 : Server | queue : ($S_2 \leftarrow 5$) $\rangle$

Looking at the code in Figure 1, one of the R/A-matches for the equivalence class of the above term is the triple $([\mathbb{C}]_A, req, [\theta]_A)$ such that

$\mathbb{C} = \odot$ $\langle$ Nt : Network | soup : ($S_1 \leftarrow 10$) $\rangle$
$\langle$ S_1 : Server | queue : nil $\rangle$ $\langle$ S_2 : Server | queue : ($S_2 \leftarrow 5$) $\rangle$

and θ is such that

$$\theta(Cl) = H, \theta(N) = 3, \theta(m_1) = 1, \theta(m_2) = 0.$$

Definition 7 (E/A-Canonical One-Step $\mathcal{R}$-Rewrite). *An E/A-canonical one-step $\mathcal{R}$-rewrite is a labelled transition of the form,*

$$[u]_A \xrightarrow{([\mathbb{C}]_A, r, [\theta]_A, [\rho]_A)} [v]_A$$

where

1. $[u]_A, [v]_A \in Can_{\Sigma, E/A}$
2. $([\mathbb{C}]_A, r, [\theta]_A)$ *is an R/A-match of* $[u]_A$
3. $[\rho]_A \in CanGSubst_{E/A}(\overrightarrow{y})$
4. $[v]_A = [\mathbb{C}(\odot \leftarrow t'(\theta(\overrightarrow{x}), \rho(\overrightarrow{y})))]_A$*, where* $\{\theta, \rho\}|_{\overrightarrow{x}} = \theta$ *and* $\{\theta, \rho\}|_{\overrightarrow{y}} = \rho$.

We associate the probability $\pi_r(\alpha_r(\rho))$ with this transition. We can now see why the $\mathcal{F}_r$ cover α_r was needed. The nondeterminism associated with the choice of the R/A-match must be removed in order to associate a probability space over the space of computations (which are infinite sequences of canonical one step $\mathcal{R}$-rewrites). The nondeterminism is removed by what is called an *adversary* of the system, which defines a probability distribution over the set of R/A-matches. In [9] a probability space is associated over the set of computation paths. To do this, an adversary for the system is fixed. We have also shown in [9] that probabilistic rewrite theories have great expressive power. They can express various known models of probabilistic systems like Continuous Time Markov Chains [8], Markov Decision Processes [10] and even Generalized Semi Markov Processes [7]. We also propose a temporal logic, to express properties of interest in probabilistic systems. The details can be found in [9].

Probabilistic rewrite theories can be used to model probabilistic actor systems [1]. Actors, which are inherently asynchronous, can be modelled naturally using object based rewriting. In probabilistic actor systems we may be interested in modelling message delay distributions among other probabilistic entities. However because time acts as a global synchronization parameter the natural encoding using objects, computing by their own rewrite rules is insufficient. The technique of *delayed* messages helps us to correctly encode time in actors. Actor failures and message drops can also be encoded. Due to space constraints we do not indicate our encoding in this paper. The file at http://maude.cs.uiuc.edu/pmaude/at.maude presents our technique.

A special subclass of **PRwTh** , called *finitary probabilistic rewrite theories*, while fairly general, are easier to implement. We describe them below.

4 Finitary Probabilistic Rewrite Theories

Observe that there are two kinds of nondeterministic choice involved in rewriting. First, the selection of the rule and second, the exact substitution-context pair. Instead of having to think of nondeterminism from both these sources, it is easier to think in terms of rewrite rules with same left-hand side term as representing the computation of some part of the system, say an object, and thus representing one nondeterministic choice. Of course the substitution and context also have to be chosen to fix a nondeterministic choice. After nondeterministically selecting a rewrite rule, instantiated with a given substitution in a given context, different probabilistic choices arise for different right-hand sides of rules having the same left-hand side as that of the chosen rule, and which can apply in the chosen context with the chosen substitution. To assign probabilities, we assign *rate* functions to rules with the same left-hand side and normalize them. The rates, which depend on the chosen substitution, correspond to the *frequency* with which the RHS's are selected. Moreover, not having new variables on the right-hand sides of rules makes the implementation much simpler. Such theories are called *finitary* probabilistic rewrite theories. We define them formally below.

Definition 8 (Finitary Probabilistic Rewrite Theory). A *finitary probabilistic rewrite theory* is a 4-tuple $\mathcal{R}_f = (\Sigma, E \cup A, R, \gamma)$, with $(\Sigma, E \cup A, R)$ a rewrite theory and $\gamma : R \to T_{\Sigma,E/A,PosRat}(X)$ a function associating to each rewrite rule in R a term $\gamma(r) \in T_{\Sigma,E/A,PosRat}(X)$, with some variables from the set X, and of sort *PosRat*, where *PosRat* is a sort in $(\Sigma, E \cup A)$ corresponding to the positive rationals. The term $\gamma(r)$ represents the *rate* function associated with rule $r \in R$. If $l : t(\overrightarrow{x}) \to t'(\overrightarrow{x})\ \mathit{if}\ C(\overrightarrow{x})$ is a rule in R involving variables $\overrightarrow{x}$, then γ maps the rule to a term of the form $\gamma_r(\overrightarrow{x})$ possibly involving some of the variables in $\overrightarrow{x}$. We then use the notation

$$l : t(\overrightarrow{x}) \to t'(\overrightarrow{x})\ \mathit{if}\ C(\overrightarrow{x})\ [\ \mathit{rate}\ \gamma_r(\overrightarrow{x})\ ]$$

for the γ-annotated rule. Notice that t' does not have any new variables. Thus, all variables in t' are also variables in t. Furthermore, we require that all rules labelled by l have the *same* left-hand side and are of the form

$$\begin{array}{c} l : t \to t'_1\ \mathit{if}\ C_1\ [\ \mathit{rate}\ \gamma_{r_1}(\overrightarrow{x})\ ] \\ \cdots \\ l : t \to t'_n\ \mathit{if}\ C_n\ [\ \mathit{rate}\ \gamma_{r_n}(\overrightarrow{x})\ ] \end{array} \tag{1}$$

where

1. $\overrightarrow{x} = \mathit{fvars}(t)\ \supseteq\ \bigcup_{1\leq i\leq n} \mathit{fvars}(t'_i) \cup \mathit{fvars}(C_i)$, that is the terms t'_i and the conditions C_i do not have any variables other than $\overrightarrow{x}$, the set of variables in t.
2. C_i is of the form $(\bigwedge_j u_{ij} = v_{ij}) \wedge (\bigwedge_k w_{ik} : s_{ik})$, that is, condition C_i is a conjunction of equations and memberships[1].

[1] The requirement $\mathit{fvars}(C_i) \subseteq \mathit{fvars}(t)$ can be relaxed by allowing new variables in C_i to be introduced in "matching equations" in the sense of [4]. Then these new variables can also appear in t'_i.

We denote the class of finitary probabilistic rewrite theories by **FPRTh**.

4.1 Semantics of Finitary Probabilistic Rewrite Theories

Given a finitary probabilistic rewrite theory $\mathcal{R}_f = (\Sigma, E \cup A, R, \gamma)$, we can express it as a probabilistic rewrite theory $\mathcal{R}_f^\bullet$, by defining a map $F_{\mathcal{R}} : \mathcal{R}_f \mapsto \mathcal{R}_f^\bullet$, with $\mathcal{R}_f^\bullet = (\Sigma^\bullet, E^\bullet \cup A, R^\bullet, \pi^\bullet)$ and $(\Sigma, E \cup A) \subseteq (\Sigma^\bullet, E^\bullet \cup A)$, in the following way. We encode each group of rules in R with label l of the form 1 above by a single probabilistic rewrite rule[2]

$$t(\overrightarrow{x}) \rightarrow \ \mathit{proj}(i, (t_1'(\overrightarrow{x}), \ldots, t_n'(\overrightarrow{x}))) \ \mathit{if} \ \widetilde{C}_1(\overrightarrow{x}) \ \mathit{or} \ \ldots \mathit{or} \ \widetilde{C}_n(\overrightarrow{x}) = \mathit{true} \\ \mathit{with\ probability}\ \pi_r(\overrightarrow{x})$$

in $R^\bullet$. Corresponding to each such rule, we add to $\Sigma^\bullet$ the sort $[1:n]$, with constants $1, \ldots, n :\rightarrow [1:n]$, and the projection operator $\mathit{proj} : \ [1:n]\ k \ldots k \rightarrow k$. We also add to $E^\bullet$ the equations $\mathit{proj}(i, t_1, \ldots, t_n) = t_i$ for each $i \in \{1, \ldots, n\}$. Note that the only new variable on the righthand side is i, and therefore $\mathit{CanGSubst}_{E/A}(i) \cong \{1, \ldots, n\}$. We consider the σ-algebra $\mathcal{P}(\{1, \ldots, n\})$ on $\{1, \ldots, n\}$. Then π_r is a function

$$\pi_r : \ [\![C]\!] \rightarrow \mathit{PFun}(\{1, \ldots, n\}, \mathcal{P}(\{1, \ldots, n\}))$$

defined as follows. If θ is such that $\widetilde{C}_1(\theta(\overrightarrow{x})) \ \mathit{or} \ \ldots \ \mathit{or} \ \widetilde{C}_n(\theta(\overrightarrow{x})) = \mathit{true}$, then $\pi_\theta = \pi_r(\theta)$ defined as

$$\pi_\theta(\{i\}) = \frac{?\gamma_{r_i}(\theta(\overrightarrow{x}))}{?\gamma_{r_1}(\theta(\overrightarrow{x})) + ?\gamma_{r_2}(\theta(\overrightarrow{x})) + \cdots + ?\gamma_{r_n}(\theta(\overrightarrow{x}))}$$

where, if $\widetilde{C}_i(\theta(\overrightarrow{x})) = \mathit{true}$, then $?\gamma_{r_i}(\theta(\overrightarrow{x})) = \gamma_{r_i}(\theta(\overrightarrow{x}))$ and $?\gamma_{r_i}(\theta(\overrightarrow{x})) = 0$ otherwise. The semantics of $\mathcal{R}_f$ computations is now defined in terms of its associated theory $\mathcal{R}_f^\bullet$ in the standard way, by choosing the singleton $\mathcal{F}$-cover $\alpha_r : \ \{1, \ldots, n\} \rightarrow \mathcal{P}(\{1, \ldots, n\})$ mapping each i to $\{i\}$.

5 The PMaude Tool

We have developed an interpreter called **PMaude**, which provides a framework for specification and execution of finitary probabilistic rewrite theories. The **PMaude** interpreter has been built on top of Maude 2.0 [4, 3] using the Full-Maude library [6]. We describe below how a finitary probabilistic rewrite theory is specified in our implemented framework and discuss some of the implementation details.

[2] By the assumption that $(\Sigma, E \cup A)$ is confluent, sort-decreasing, and terminating modulo A, and by a metatheorem of Bergstra and Tucker, any condition C of the form $(\bigwedge_i u_i = v_i \wedge \bigwedge_j w_j : s_j)$ can be replaced in an appropriate protecting enrichment $(\widetilde{\Sigma}, \widetilde{E} \cup A)$ of $(\Sigma, E \cup A)$ by a semantically equivalent Boolean condition $\widetilde{C} = \mathit{true}$.

Consider a finitary probabilistic rewrite theory with k distinct rewrite labels and with n_i rewrite rules for the i^{th} distinct label, for $i = 1, 2, \dots, k$.

$$\begin{array}{l}
l_1 : t_1 \rightarrow t'_{11} \quad \mathit{if}\ C_{11}\ [\ \mathit{rate}\ \gamma_{11}(\overrightarrow{x})\ ] \\
\quad \dots \\
l_1 : t_1 \rightarrow t'_{1n_1}\ \mathit{if}\ C_{1n_1}\ [\ \mathit{rate}\ \gamma_{1n_1}(\overrightarrow{x})\ ] \\
\quad \dots \\
l_k : t_k \rightarrow t'_{k1}\ \mathit{if}\ C_{k1}\ [\ \mathit{rate}\ \gamma_{k1}(\overrightarrow{x})\ ] \\
\quad \dots \\
l_k : t_k \rightarrow t'_{kn_k}\ \mathit{if}\ C_{kn_k}\ [\ \mathit{rate}\ \gamma_{kn_k}(\overrightarrow{x})\ ]
\end{array}$$

At one level we want all rewrite rules in the specification to have *distinct* labels, so that we have low level control over these rules, while at the conceptual level, groups of rules must have the same label. We achieve this by giving two labels: one, common to a group and corresponding to the group's label l at the beginning, and another, unique for each rule, at the end. The above finitary probabilistic rewrite theory can be specified as follows in **PMaude**.

pmod FINITARY-EXAMPLE is
  cprl $[l_1] : t_1 \Rightarrow t'_{11}$ *if* C_{11} [*rate* $\gamma_{11}(x_1, \dots)$] [*metadata* "l_{11} ..."] .
  ...
  cprl $[l_1] : t_1 \Rightarrow t'_{1n_1}$ *if* C_{1n_1} [*rate* $\gamma_{1n_1}(x_1, \dots)$] [*metadata* "l_{1n_1} ..."] .
  ...
  cprl $[l_k] : t_k \Rightarrow t'_{k1}$ *if* C_{k1} [*rate* $\gamma_{k1}(x_1, \dots)$] [*metadata* "l_{k1} ..."] .
  ...
  cprl $[l_k] : t_k \Rightarrow t'_{kn_k}$ *if* C_{kn_k} [*rate* $\gamma_{kn_k}(x_1, \dots)$] [*metadata* "l_{kn_k} ..."] .
endpm

User input and output are supported as in Full Maude using the `LOOP-MODE` module. **PMaude** extends the Full Maude functions for parsing modules and any terms entered later by the user for rewriting purposes. Currently **PMaude** supports four user commands. Two of these are low level commands used to change seeds of pseudo-random generators. We shall not describe the implementation of those two commands here. The other two commands are rewrite commands. Their syntax is as follows:

(*prew* t .)
(*prew*-$[n]$ t .)

The default module M in which these commands are interpreted is the last read probabilistic module. The *prew* command is an instruction to the interpreter to probabilistically rewrite the term t in the default module M, till no further rewrites are possible. Notice that this command may fail to terminate. The *prew*-$[n]$ command takes a natural number n specifying the maximum number of probabilistic rewrites to perform on the term t. This command always terminates in at most n steps of rewriting. Both commands report the final term (if *prew* terminates).

The implementation of these commands is as follows. When the interpreter is given one of these commands, all possible one-step rewrites for t in the default module M are computed. Out of all possible groups $l_1, l_2, .., l_k$ in which *some* rewrite rule applies, one is chosen, *uniformly at random.* For the chosen group l_i, all the rewrite rules $l_{i1}, l_{i2}, .., l_{in_i}$ associated with l_i, are guaranteed to have the same left-hand side $t_i(x_1, x_2, ..)$. From all possible canonical substitution, context pairs $([\theta]_A, [\mathbb{C}]_A)$ for the variables x_j, representing successful matches of $t_i(x_1, x_2, ..)$ with the given term t, that is, matches that also satisfy one of the conditions C_{ij}, one of the matches is chosen *uniformly at random.* The two steps above also define the exact adversary we associate to a given finitary probabilistic rewrite theory in our implementation. If there are m groups, $l_{i_1}, \ldots, l_{i_m}$, in which *some* rule applies and v_j matches in total for group l_{i_j} then the adversary chooses a match in group l_{i_j} with probability $\frac{1}{mv_j}$. To choose the exact rewrite rule l_{ij} to apply, use of the rate functions is made. The values of the various rates γ_{ip} are calculated for those rules l_{ip} such that $[\theta]_A$ satisfies the condition of the rule l_{ip}. Then these rates are normalized and the choice of the rule l_{ij} is made *probabilistically*, based on the calculated rates. This rewrite rule is then applied to the term t, in the chosen context with the chosen substitution. If the interpreter finds no successful matches for a given term, or if it has completed the maximum number of rewrites specified, it immediately reports that term as the answer. Since rates can depend on the substitution, this allows users to specify systems where probabilities are determined by the state.

PMaude can be used as a simulator for finitary probabilistic rewrite theories. The programmer must supply the system specification as a **PMaude** module and a start term to rewrite. To obtain different results the seeds for the random number generators must be changed at each invocation. This can be done by using a scripting language to call the interpreter repeatedly but with different seeds before each execution.

We have specified the client-server example discussed in Section 3 in **PMaude** with the following parameters: The client only sends two kinds of packets, loads 5 and 10, with equal probability. The request messages for S_1, S_2 are dropped with probabilities 2/7 and 1/6 respectively, while acknowledgement messages for S_1, S_2 are dropped with probabilities 1/6, 1/11 respectively. We also chose S_1 to drop processing of a request with probability 1/7 as opposed to 3/23 when its load was at least 100, while for S_2 the load limit was 40 but the probabilities of dropping requests were 1/7 and 3/19. We performed some simulations and, after a number of runs, we computed the ratio $(svc_1 + svc_2)/sent$ as a measure of the quality of service for the client. Simulations showed that among static policies, namely those where the client did not adapt sending probabilities on the fly, that of *sending twice as often* to server S_2 than to S_1 was better than most others.

The complete code for the **PMaude** interpreter, as well as several other example files, can be found at http:/maude.cs.uiuc.edu/pmaude/pmaude.html.

6 Conclusions and Future Work

Probabilistic rewrite theories provide a general semantic framework supporting high level probabilistic specification of systems; in fact, we have shown how various well known probabilistic models can be expressed in our framework [9]. The present work shows how our framework applies to concurrent object based systems. For a fairly general subclass, namely finitary probabilistic rewrite theories, we have implemented a simulator **PMaude** and have exercised it on some simple examples. We are currently carrying out more case studies. We have also identified several aspects of the theory and the **PMaude** tool that need further development.

On the more theoretical side, we feel research is needed in three areas. First, it is important to develop a general model of probabilistic systems with *concurrent* probabilistic actions, as opposed to the current *interleaving* semantics. Second, deductive and analytic methods for property verification of probabilistic systems, based on our current framework is an important open problem. Algorithms to translate appropriate subclasses to appropriate representations enabling use of existing model checkers, should be developed and implemented.

Third, we think that allowing the probability function π_r to depend not only on the substitution, but also on the context would give us more modelling power. Specifically, it would enable us to represent applications where the probability distributions of certain variables, such as message delays, depend on functions of the entire state of the system, for example, on the congestion in the network. Such situations can also be modelled in our current framework but at the expense of using rewrite rules at the *top-level*, whose substitutions capture global system parameters. Such rules can modify the entire system at once as opposed to just modifying local fragments, but at the cost of violating the modularity principle of concurrent objects or actors.

On the implementation side, an extension of the **PMaude** framework to enable specification of more general classes of probabilistic rewrite theories and adversaries is highly desirable. This will allow the generation of simulation traces for the system under consideration and can be used as a tool to implement the model independent Monte-Carlo simulation and acceptance sampling methods for probabilistic validation of properties [14]. As an application of our theory, we believe that it will be fruitful to model networked embedded systems, where apart from time, there are other continuous state variables, such as battery power or temperature, whose behavior may be stochastic. Moreover, the properties of interest are often a statistical aggregation of many observations.

Acknowledgement

The work is supported in part by the Defense Advanced Research Projects Agency (the DARPA IPTO TASK Program, contract number F30602-00-2-0586 and the DARPA IXO NEST Program, contract number F33615-01-C-1907) and the ONR Grant N00014-02-1-0715. We would like to thank Wooyoung Kim for reading a previous version of this paper and giving us valuable feedback and Joost-Pieter Katoen for very helpful discussions and pointers to references.

References

1. G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.
2. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236(1–2):35–132, 2000.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Towards maude 2.0. In K. Futatsugi, editor, *Electronic Notes in Theoretical Computer Science*, volume 36. Elsevier, 2001.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude 2.0 Manual, Version 1.0*, june 2003. `http://maude.cs.uiuc.edu/manual/maude-manual.pdf`.
6. F. Durán and J. Meseguer. Parameterized theories and views in full maude 2.0. In K. Futatsugi, editor, *Electronic Notes in Theoretical Computer Science*, volume 36. Elsevier, 2001.
7. P. Glynn. The role of generalized semi-Markov processes in simulation output analysis, 1983.
8. J.-P. Katoen, M. Kwiatkowska, G. Norman, and D. Parker. Faster and symbolic CTMC model checking. *Lecture Notes in Computer Science*, 2165, 2001.
9. N. Kumar, K. Sen, J. Meseguer, and G. Agha. Probabilistic rewrite theories: Unifying models, logics and tools. Technical Report UIUCDCS-R-2003-2347, University of Illinois at Urbana-Champaign, May 2003.
10. M. Z. Kwiatkowska, G. Norman, and D. Parker. Prism: Probabilistic symbolic model checker, 2002.
11. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
12. J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.
13. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, pages 18–61. Springer LNCS 1376, 1998.
14. H. L. S. Younes and R. G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 223–235, Copenhagen, Denmark, July 2002. Springer.

Engineering the SDL Formal Language Definition

Andreas Prinz[1] and Martin v. Löwis[2]

[1] DResearch Digital Media Systems GmbH, Otto-Schmirgal-Str. 3,
D-10319 Berlin
`prinz@dresearch.de`

[2] Hasso-Plattner-Institut für Softwaresystemtechnik GmbH, Postfach 900460,
D-14440 Potsdam
`martin.vonloewis@hpi.uni-potsdam.de`

Abstract. With the latest revision of the ITU-T Specification and Description Language (SDL-2000), a formal language definition based on the concept of Abstract State Machines (ASMs) became integral part of the standard. Together with the formal definition, we have developed software tools that allow executing the formal language definition on a computer. In doing so, we found that tools greatly help to eliminate numerous errors from the formal definition, which likely would have not been found without tools.

1 Introduction

Defining the formal semantics of programming languages is an area of ongoing research. While formalizing the syntax of the language by means of grammars is well-understood, various techniques have been studied to define other properties of the language. We think that a language is sufficiently formalized if the following aspects are defined [1]:

1. The lexis: How to transform the input sequence of bytes into a sequence of tokens?
2. The syntax: Given a sequence of tokens, is that a sentence of the language?
3. The well-formedness rules (static semantics): Given a sentence of the language, does that meet certain additional constraints?
4. The run-time (or dynamic) semantics: What does the program do when executed?

In this paper, we discuss these aspects of the language definition for the ITU-T Specification and Description Language [2]. We first introduce the language SDL, and continue with an overview of the formal semantics, covering each of the four aspects.

In our experience, defining a formal semantics for a language is not useful without sufficient tool support. Tools should be used to verify that the formal definition is consistent in itself, and to validate it agrees with the intended semantics of the language. Unfortunately, no single tool can be used to automatically process a formal language definition from the version that is published as an international standard. Consequently, we have developed our own tool chain (building on existing tools where possible) to process the formal semantics definition of SDL. The tools and the results we achieved by using the tools are presented in subsequent sections. We hope to expose this approach to a wider audience, and to encourage researchers to copy either the general approach or the choice of specific tools for their language of interest.

The SDL formal semantics was developed by an international expert team including the authors of this paper[1]. Due to this effort SDL-2000 does now have a complete semantics for the full language, not only for a subset of it. This semantics is precise for all aspects of the language including syntax and semantics.

Success of the development of a formal semantics for a programming language depends on many aspects, of which this paper covers only a few. In particular, we believe that standardization of the formal language definition is quite important: If the formal language definition is *the* official semantics of the language, practitioners will use the formal semantics for their own work. For SDL-2000, we managed to convince the SDL standards body (ITU-T Study Group 10) to accept the formal semantics presented in this paper as the official semantics definition of SDL.

2 The Language SDL-2000

SDL (Specification and Description Language) is a language for the description of reactive systems [2]. Although developed for the telecom industry, SDL is used in other areas of the industry, as well. In the area of telecommunications, SDL can be used for the following application fields: call and connection processing, maintenance and fault treatment, system control, operation and maintenance functions, network management, data communication protocols, and telecommunication services[2].
More generally, SDL can be used for the functional description of all systems that can be described with a discrete model; that is where the object communicates with its environment by discrete messages [2]. Core concepts to describe such systems are:

- agents, which are used to describe active objects which respond to external stimuli;
- channels, which connect agents;
- signals, which are transmitted via channels;
- data, which describe the state of agents and the parameters of signals.

SDL features both a graphical and a textual syntax. While the graphical syntax allows human readers to grasp the structure of the system and the behaviour of an agent more easily, a machine more easily processes the textual syntax.

3 The SDL Formal Semantics

In November 2000, the formal semantics of SDL-2000 was officially approved to become part of the SDL language definition [3]. Currently, it is the only comprehensive and complete formal semantics of SDL-2000, covering all static and dynamic

[1] It should be noted that the SDL formal semantics was developed in only one years time, which is really fast taking into account the size of the informal language description (without examples and indexes) taking 350 pages.

[2] SDL is a language with a long tradition. It was first defined in 1976 and evolved since then to a very complex language. It is standardised by the International Telecommunication Union (ITU-T) by the standard Z.100. Since 1988 there is a formal semantics standardised together with the (informal) language definition. The SDL-2000 formal semantics was written in the ASM formalism ([1], [6]) in contrast to the 1988 formalisations written in Meta IV and CCS.

language aspects. From the beginning it was agreed to define the SDL formal semantics to be truly executable. In this section, the static and dynamic semantics of SDL-2000 are surveyed. Further details on the dynamic semantics and, in particular, on the use of the ASM formalism can be found in [6].

3.1 Static Semantics

The static semantics covers transformations and checks that can be done before executing a specification. In the scope of SDL, there are two major parts of the static semantics:

- *Well-formedness conditions*: As usual, the SDL concrete syntax is given in a context-free way. Additional constraints are imposed using context conditions.
- *Transformations*: In order to cope with the complexity of the language SDL, the standard Z.100 identifies certain concepts to be core concepts and defines transformations of various other concepts into these core concepts.

Starting point for defining the static semantics of SDL is a syntactically correct SDL specification as determined by the SDL grammar. In Z.100, a concrete textual, a concrete graphical, and an abstract grammar are defined using Backus-Naur-Form (BNF) with extensions to capture the graphical language constructs. From a syntactically correct SDL specification, an abstract syntax tree (AST) is derived by standard compiler techniques (namely, parser construction for a context-free grammar). The structure of this AST is defined such that it resembles the concrete textual and the concrete graphical grammars. The correspondence between the concrete grammars and a first abstract syntax form, called AS0, is almost one-to-one, and removes irrelevant details such as separators and other 'purely' lexical information. A second step translating AS0 to the final abstract syntax form, called AS1, is formally captured by a set of transformation rules together with a mapping. This results in the following structure of the formalization (see figure 1):

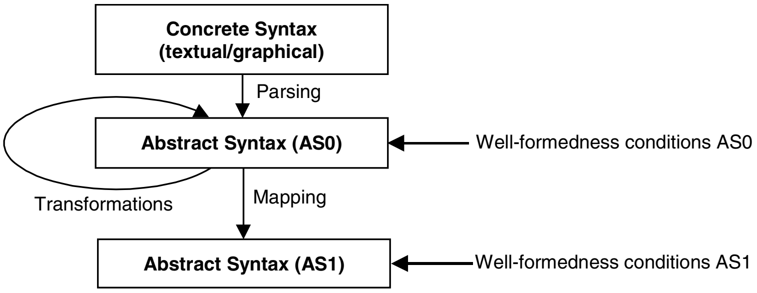

Concrete Syntax (textual/graphical)
Parsing
Abstract Syntax (AS0)
Well-formedness conditions AS0
Transformations
Mapping
Abstract Syntax (AS1)
Well-formedness conditions AS1

Fig. 1. Overview of the static semantics

- Transformation rules modifying AS0 trees are described in so-called *model paragraphs* of Z.100, and are formally expressed as rewrite rules.
- After application of the transformations, the structure of the AS0 tree is similar to an AS1 tree. This means that the mapping from AS0 to AS1 is almost one-to-one.
- The well-formedness conditions are split into conditions on AS0 and AS1 (see figure 1). They are formalized in terms of first-order predicate calculus.

Transformations. Z.100 prescribes the transformation of SDL specifications by a sequence of *transformation steps*. Each transformation step consists of a set of single transformations as stated in the *Model* sections of Z.100, and determines how to handle one special class of shorthand notations, i.e., abbreviations. The result of one step is used as input for the next step. The following excerpt of Z.100 prescribes how to deal with a particular shorthand notation, namely a list of output signals:

> If several <signal identifier>s are specified in an <output body>, this is derived syntax for specifying a sequence of <output>s in the same order as specified in the original <output body>.

The rule states that a list of output signals is a shorthand (called "derived syntax") for a sequence of single outputs.

To formalize the transformation rules of Z.100, rewrite rules are used. A single transformation is realized by the application of a rewrite rule to the specification, which essentially means to replace parts of the specification by other parts as defined by the rule. The following rewrite rule formalizes the transformation above:

```
< <output>(<output body>(< o > //  r) ) > provided r ≠ empty
=1=>
< <output>(<output body>(< o >) ), <output>(<output body>(r) ) >
```

The left hand side (above the arrow) gives the application context of the rule, i.e. an output body containing more than one <signal identifier>, in the abstract syntax tree. The right hand side defines how this part of the AS0 tree is to be transformed. The number in the operator =n=> indicates the sequence of transformation steps. All the rules are applied as long as they are enabled. For instance, if the sequence of signal identifiers has 3 elements, two applications of the previous rule are required.

Well-Formedness Conditions. The *well-formedness conditions* define additional constraints that a specification has to satisfy. These constraints cannot be expressed in BNF, though they are static and can be defined and checked independently of the dynamic semantics definition. An SDL specification is *valid* if and only if it satisfies the syntactical rules and the static conditions of SDL.

Here is a sample resolution rule taken from Z.100, and its formalization:

> The <signal identifier> in an <output body> must denote a <signal definition>.
> ∀o ∈ <output body>: o.**s-**<signal identifier>.refersto$_0$ ∈ <signal definition>

There is again an application context for this rule given after the ∀ symbol – the rule is applicable for <output body> AS0 nodes. The <signal identifier> selected from this node must refer to a <signal definition>. The actual resolution is done by the function refersto$_0$, which is formally defined in [3] covering about four pages.

3.2 Dynamic Semantics

The *dynamic semantics* defines the dynamic properties resulting from the execution of valid SDL specifications, i.e., their legal behavior. It is based on Abstract State Machines (ASMs) introduced by Yuri Gurevich [5], and defines, for each SDL specification, a *distributed real-time ASM*, covering concurrency, asynchronicity, and time (see figure 2)[3].

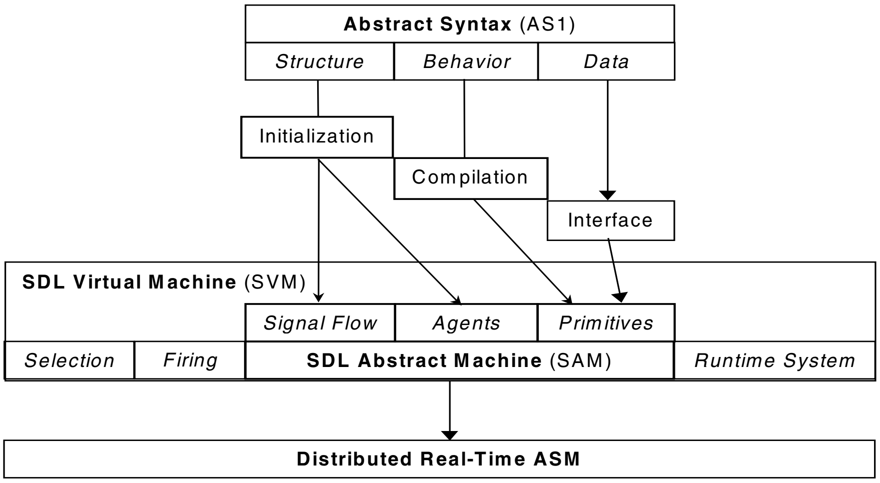


Fig. 2. Overview of the dynamic semantics

The core of our model is a logical hardware called *SDL Abstract Machine* (*SAM*), providing a dynamic architecture according to the SDL specification under execution, various types of agents, and an instruction set. Each SDL specification is mapped by the formal semantics to the SAM. The execution of the specification is then performed within the SAM using an abstract operating system called SVM (SDL virtual machine).

SDL Virtual Machine. The SDL Virtual Machine, or SVM, has the SDL Abstract Machine, or SAM, as its heart. Essentially, the SVM adds a run-time system to the SAM, which consists of the programs to drive the execution of the behavior primitives[4], and the run-time libraries used within the behavior primitives. The SVM consists of fixed parts, which are independent of the actual SDL specification, and dynamic parts created by traversing the abstract syntax[5]. The SAM consists of the following parts (see figure 2):

- *Signal Flow Model*, which defines a uniform treatment of signal flow related aspects, in particular, asynchronous communication between agents through exchange of signals via channels connected to gates,
- *SAM Agents*, which model the SDL concepts 'SDL agent', 'SDL agents set', and 'SDL channel', and
- *Behavior Primitives*, which can be seen as the instructions of the SAM.

---

[3] This means the SDL dynamic semantics is an operational one which is based on state transitions in terms of abstract state machines. Please note that no inference or rule system is involved in this kind of semantics definition (as opposed to a denotational semantics).

[4] the selection and firing code.

[5] both in the process of behavior compilation, and by recursively unfolding the system structure.

The signal flow model is based on a decentralized control mechanism for defining asynchronous communication between SDL agents (such as processes, blocks or a system). Basically, it deals with the transportation of *signals* between (SDL) agents via their *gates*. Exchange of signals between SDL agents and their environment is modeled by means of *gates* from a static domain *GATE*. A gate forms an interface for *serial* and *unidirectional* communication between two or more agents. Accordingly, gates are either classified as *input gates* or *output gates.*

A static domain *SIGNAL* represents the set of all signal types as declared by an SDL specification. Dynamically created signal instances belong to a dynamic domain *SIGNALINST*. Basic functions on signals are *signalSender*, *toArg*, and *viaArg* yielding the sender process, the destination, and optional constraints on admissible communication paths, respectively.

SAM Agents. A distributed ASM defines an *asynchronous* computation model consisting of some finite collection of autonomously operating agents. Each agent executes a program, where agents that execute the same program are considered to be of the same type. Even SDL connections are modeled as agents executing the program LINKPROGRAM[6]. Agents interact with each other by reading and writing shared locations of global machine states. In the case of the SDL semantics these shared locations are the gates. The underlying semantic model resolves potential conflicts by making nondeterministic choices according to the definition of *partially ordered* machine runs [5].

The behavior of a link is stated by the state transition rule FORWARDSIGNAL forming the program LINKPROGRAM. For improved readability of ASM programs, complex transition rules are structured by means of rule macros that often have formal parameters. In the definition shown below of the rule FORWARDSIGNAL, *Self* identifies a particular link agent (the one executing the rule).
A link agent *l* basically performs a single operation, namely: signals received at gate *l.from* are forwarded to gate *l.to*. Whenever *l* is applicable to a waiting signal *si* (as identified by the *l.from.queue.head*), it removes *si* from *l.from.queue* and inserts it into *l.to.schedule*. Competing attempts of two or more link agents to forward the same signal *si* cannot cause a duplication of the signal. Technically, this property is ensured by the underlying concurrency model (cf. the coherence condition in the definition of partially ordered runs [5]).

```
FORWARDSIGNAL ≡
    if Self.from.queue ≠ empty then
        let si = Self.from.queue.head in
            if Applicable(si.signalType,si.toArg,si.viaArg,Self.from,Self) then
                DELETE(si,Self.from)
                INSERT(si,now+Self.delay,Self.to)
```

[6] Note that the resulting signal flow model architecture is fairly robust allowing for the incorporation of additional features in future versions of SDL. For instance, one may have channels with more complex properties (like unreliable transmission behavior) and a dynamically changing communication infrastructure (with channels being added and removed at run time). Such extensions can be easily expressed on the basis of the decentralized signal flow model without any major revision of the current definitions.

```
                si.viaArg := si.viaArg \ { Self.from.nodeAS1.node.AS1ToId,
                                          Self.nodeAS1.node.AS1ToId}
            endif
        endlet
    endif
```

Links are just a special case of agents connected via gates. In the SDL formal semantics, there are also process set agents and process agents. The behaviour of process set agents is similar to the link behaviour. The behaviour of process agents is given by the SDL specification and transformed to behaviour primitives as described below.

Behaviour Primitives. The behaviour described in the SDL specification is transformed by the formal semantics to a set of behaviour primitives, forming the machine code of the SAM. This transformation is formalised using a compilation function. The primitives are fairly high-level and correspond to the possible SDL actions. This way, the formal semantics is working on abstract code at runtime and not on the abstract syntax tree.

4 The Development Process of the Formal Language Definition

The SDL language specification consists of several parts. Some of these are primarily informal, and given in plain English; other parts are primarily written in formal languages:

- The main part of the language definition in the ITU recommendation Z.100 [2] defines the syntax of SDL, by providing an EBNF (extended Backus-Naur-Form) grammar; it defines the semantics of SDL using English text.
- Annex A is the index of the non-terminal symbols of the grammar.
- Annex D defines the predefined data (package predefined) using a formal syntax. The semantics of the predefined data is given informally in this annex.
- Annex F [3] defines the formal semantics. This annex consists of three parts: F.1 (overview of the formal semantics), F.2 (static semantics) and F.3 (dynamic semantics).

Different working groups developed these documents over a period of several years. They were working on the informal and the formal definition in parallel. In order to reach a consistent language definition, a development process was followed to systematically formalise the completed parts of the informal definition.

Due to the groups involved, and the documents produced, the development process had several phases, as shown in figure 3.

5 The Tool Chain

Two main approaches lead to the discovery of inconsistencies and other problems in the language definition. On the one hand, the mere attempt to formalise a certain concept lead to the observation, that certain constellations have no clear meaning. In other words, we found some errors by "just looking" at the specification[7].

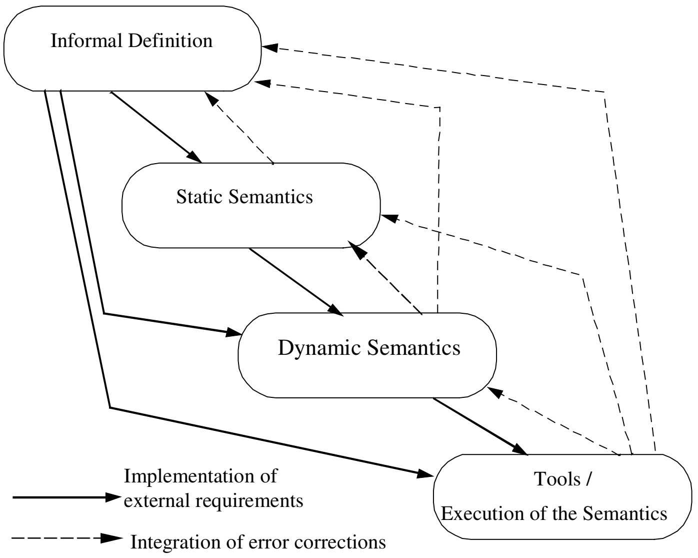

Informal Definition
Static Semantics
Dynamic Semantics
Tools /
Execution of the Semantics
Implementation of external requirements
Integration of error corrections

Fig. 3. Development process of the formal semantics definition

On the other hand, various problems where detected using software tools. Figure 4 gives an overview of these tools. The main idea of the tools is to use the formal semantics as a high-level description of an SDL compiler, which is turned into a lower-level description using a transformation tools called SDLC. However, also the lower-level description is still on a high level such that the transformation does not need to introduce major changes to the description[8].

As required by the ITU-T administration, the formal semantics documents had to be maintained using the Microsoft Word [9] tool. To define the formal semantics, the following features of that tool were used:

- Using text labels and hypertext links, easy navigation in the document becomes possible. For example, every call to a function refers (as a hyperlink) to the function definition.

[7] Even though this approach is hardly systematic and trustworthy, it is quite reliable if executed carefully: If, for example, the language designer finds that the formalisation of an algorithm requires data that are unavailable to that algorithm, the language designer will try to solve this by extending the formal language definition with the necessary data. In turn, he might find that this extension cannot be implemented, and that the informal definition needs to specify a different algorithm.

[8] Please note that there are two transformations to be considered. The first transformation is from the formal semantics definition of SDL to the SDL reference compiler, shown in the left part of figure 4. The second transformation is from a concrete SDL specification to an output trace of this specification, which is shown in the right part of the figure.

- Using auto correction entries, entering these links is simplified. For example, entering the string \gdecision results in a hyperlink to the grammar rule *Decision*.
- Using Visual Basic macros, entering the auto correction entries is simplified. One such macro searches the entire document for grammar rules, function definitions, and other definitions, and creates the necessary auto correction entries.
- Using character and paragraph formats, both the generator for the auto correction entries, and the extraction process is driven.

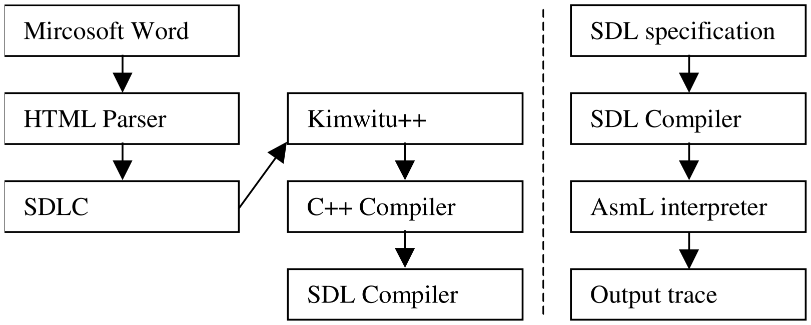


Fig. 4. The Tool Chain

The Word document has, alternating, formal and informal parts. The informal parts describe the desired operation of the formulae. Although very important to the reading human, they are not needed to execute the formal language definition. The formal parts are not easy to process as long as they are stored in the internal Microsoft Word format. To allow further processing of the formal definition, it is first converted into HTML using the HTML generator built into Word 2000, which is then processed with an HTML parser to extract the formulae.

Even though it may appear that the chosen approach of transformation steps is tied to the specific input format (i.e. Microsoft Word), it would be possible to adapt the process to different input formats. For example, if the input was available as a PostScript file, the HTML parser would need to be replaced with a PostScript parser.

These extracted files are then compiled using the compiler SDLC (developed in parallel to the language definition) into Kimwitu++ code [4, 8]. Kimwitu++ compiles the code into C++ code. A C++ compiler creates then an executable program from the C++ code.

The resulting executable program is an SDL reference compiler automatically derived from the language definition. It takes an SDL specification as its input and produces a set of AsmL [7] fragments. Those are combined with other AsmL fragments (extracted from the Word document). The AsmL compiler translates all these fragments into a C# program, which the C# compiler [10] compiles into a Microsoft .NET executable program.

Executing this program produces a possible trace of the original SDL specification.

Nearly all of these translation steps can fail. We will demonstrate a few typical problems found using the tools, and how they can be corrected.

6 Errors Detected by SDLC

If the SDLC compiler detects errors, they indicate problems in the formulae extracted from the word document. Several such errors are shown in the following sections.

6.1 Bugs in the Extraction Tool

When processing the HTML generated with Microsoft Word, specific aspects of that HTML format had to be considered, in particular mathematical characters: The formulae of the SDL language definition make use of special characters (such as ∃ and ∀). These are represented using the symbol font in HTML, whereas the extracted form expected by SDLC uses an ASCII notation (\exists and \forall) for these characters. Each such character has to be specifically added to the extraction tool. It took some time to enable the extraction tool to correctly extract the formal texts.

6.2 Formatting Errors in the Word Document

The compiler SDLC uses different namespaces for functions, domains, ASM rules, and ASM variables. Those need to be marked up in the Word document with a certain character format, and are converted to prefixes (f-, d-, r-, and a-) during extraction. The extractor produces those prefixes every time the <span> element is encountered in the HTML file. Using this strategy, a number of problems have been found:

- Incorrect mark-up: Sometimes, not only the identifier was marked, but also the white space around it. The extraction tool would then put the prefix in front of the white space, which caused SDLC to report a syntax error. This error needs to be corrected by removing the formatting of the white space.
- Missing mark-up: Some identifiers where not appropriately marked, in particular, if entering the proper mark-up was too tedious for the author of the formal semantics. In those cases, the proper mark-up needed to be added during revision.

6.3 Syntax Errors

Passing the extracted files to SDLC for the first time, numerous syntax errors were reported which were not just the result of incorrect formatting. Those errors can be classified in the following categories:

- Systematic usage of incorrect syntax: A number of errors happened as authors of the formal semantics were not always aware of the syntax to be used in the formulae. Using the formulae language just from memory, they naturally made errors in applying the syntax properly. One example is the usage of the **case** expression. A proper usage example of this expression reads

  case d' **of**
  | TYPEDEFINITION$_1$\ *Interface-definition* => *d* = *getEntityDefinition$_1$*(*d'*.**s-***Identifier*, *d'*.*entityKind$_1$*)
  | *Interface-definition* =>
  $d \in$ *Interface-definition* $\wedge$
  ($\exists$ *dataId* $\in$ *Data-type-identifier*: *dataId.parentAS1*= *d'* $\wedge$
  d = *getEntityDefinition$_1$*(*dataId*, **sort**))

```
| Syntype-definition =>
        $isDirectSuperType_1(d,d'.derivedDataType_1)$
**otherwise** False
**endcase**
```

In some cases, this syntax was misspelled: Instead of using **of** as the keyword, the word **in** was used, and instead of using "=>" as the separator between the pattern and the result expression, the separator ":" was used. Those errors were detected when passing the input to SDLC, and had to be corrected in the Word document.

- Mismatching parentheses: A large number of missing closing parentheses were found; there were also closing parentheses without a matching opening parenthesis. As the formulae language has several different parentheses kinds ("()", "{}", and "<>"), some of these errors resulted from incorrectly swapping closing parentheses.
- Unsupported constructs: A number of syntax errors were caused by using a syntax in a formula which was not supported in SDLC. We could eliminate some of those errors by enhancing SDLC (and thus enhancing the language for formulae). In other cases, the formula had to be rewritten to use only supported constructs[9].

6.4 Errors in the SDLC Output

After successfully processing the formal definition with SDLC producing Kimwitu++ code, we found that the resulting Kimwitu++ code was incorrect, and rejected either by Kimwitu++, or by the C++ compiler. Those errors can be classified as follows:

- Type errors originating from SDLC: In a number of cases, the resulting C++ code showed type errors, even though the formal SDL definition was correct. Some of these errors resulted from an incomplete type analysis inside SDLC: SDLC tries to translate the formulae on a syntactic level only, without performing a complete static analysis. This turned out to be insufficient in some cases, e.g. when the compiler needs to declare the type of a variable in C++, when no such declaration is necessary in the formula. SDLC then uses a heuristics to output a C++ type name; this heuristics turned out to be incorrect. We have then corrected the generator to improve the heuristics, using automatic type conversion available in C++.
- Incorrect arguments to function calls: In the static semantics of SDL, a specification is represented as a tree of a certain grammar. In order to manipulate the tree, new nodes in that tree have to be created, which is expressed as a function call of the non-terminal to be created; the arguments to the function call are the subtrees appearing on the right-hand side of the grammar production. These calls are nearly-literally copied into the C++ code, so that the C++ compiler would verify the correct usage of the non-terminal. In a number of cases, there were arguments missing, or extra arguments given. We found that in most of these cases, the error resulted from a change to the grammar that was inconsistently applied (i.e. ignoring all occurrences of the non-terminal that was changed).

[9] For example, to denote a sequence domain with elements originating from a domain D, the construct D* is used. To denote an optional value (i.e. a domain that also includes the value *undefined*), [D] is used. Even though SDLC separates both constructs, it rejects the construct [D]*, i.e. a list of optional values. We enhanced SDLC to support this notation as well.

7 Executing the Static Semantics

After correcting the errors described in section 6, a compiler translating SDL to AsmL can be generated and an SDL specification can be passed to the generated reference compiler. This, in turn, can produce various errors, which are shown in the following.

7.1 Checking the Well-Formedness Conditions

As the first step in processing an SDL specification, a number of well-formedness conditions (expressed in the predicate calculus) are checked for the input. These checks can fail for a number of reasons:

- Errors in the SDL specification: In the ideal case, a failed condition indicates a true error in the SDL specification. In that case, further processing of the SDL specification is meaningless, and the specification needs to be revised.
- Errors in formalising the well-formedness condition: It is possible that the SDL specification is correct, and that the informal language definition says it is correct, but an error was made when formalising the condition. In that case, the formula of the condition needs to be corrected.
- Errors in the informal definition: A well-formedness predicate may fail if the informal language definition is erroneous or too strong. In this case, the language committee experts need to determine what the intended condition was, and find proper language to describe it. Afterwards, the formalisation needs to be updated.

In addition to conditions that fail, it is possible that a condition erroneously holds, or that its computation does not terminate. The former case is difficult to detect: One needs an SDL specification that is known to be ill-formed, and then find that this specification is not rejected in the reference compiler. Once the problem is detected, it may, again, either need to be fixed in just the formal language definition, or in the informal one as well.

On the other hand, finding a condition that does not terminate is relatively easy: If the compilation of a specification in the reference compiler does not terminate in a reasonable time, the likely cause is an error in some formula. For many constructs, it is obvious that they terminate if all their arguments terminate. We found only two cases of infinite algorithms in the formal semantics, namely infinite recursion[10] and quantifications over infinite domains[11].

7.2 Execution of Transformation Rules

In SDL, a number of constructs are defined as short cuts for other constructs. They don't have a dynamic semantics of their own, but receive their semantics by means of

[10] In order to compute some data, auxiliary functions were used that ultimately invoked the same functions that called them, passing the same arguments that were currently under computation. In these cases, a different algorithm had to be found.

[11] Using quantifications is common to specify well-formedness conditions. Mostly it is obvious that the computation of them will terminate, for instance many of the quantifications run over syntax nodes, and in any specification, there is only a finite number of syntax nodes. However, we found quantifications over the domain of natural numbers, which had to be rewritten to operate only on a finite subset of all numbers.

transformation to other constructs. Those transformations are formalized by means of a rewrite system (executed by an abstract state machine) [1][12]. In executing the transformations, we found problems with infinite loops[13] and skipped transformations[14]:

7.3 Generating AsmL

Once transformations are complete, the resulting syntax tree was compiled into a sequence of AsmL statements, which were then passed to the AsmL compiler. This compiler detected further errors, which can be classified as follows:

- Constructs unsupported in AsmL: The initial release of AsmL did not support a number of constructs needed for executing SDL specifications. In turn, we forbeared usage of AsmL version 1.5, and switched to a beta release of AsmL version 2, which does support the constructs we need.
- Type errors detected by AsmL: The AsmL compiler reported a number of type errors. They mainly resulted from AsmL using a slightly different typing system than the SDL formal semantics, especially as far as *undefined* is concerned. Fortunately, only a few formulae made use of this typing, and we could eliminate those with additional type declarations in a few places.
- Bugs in AsmL: The initial beta release of AsmL would crash on the generated ASM. Together with the authors of AsmL, we could eliminate these problems.
- Inconsistencies in the semantics definition: Some inconsistencies show up as inconsistent update sets within AsmL. This means that one location gets two different values assigned. This rare kind of errors was easy to correct.
- Ambiguities in the semantics definition: Sometimes there were competing update sets within AsmL. Some of them relate to nondeterminism within SDL (e.g. channels competing for a signal, which could be relayed via different paths). However, some ambiguities related to weaknesses in the formal semantics.

8 Executing the Dynamic Semantics

The AsmL compiler (in the .NET revision) takes the generated AsmL input, together with a few runtime functions, and produces a C# program. The C# compiler then generates a .NET executable, which can be run using the .NET runtime environment.

[12] Each rewrite rule is defined as a pattern over a subtree of the syntax tree which needs to match for the rule to fire, and a result term which determines the syntax subtree that is used as a replacement for the original subtree.

[13] Some of the rewrite rules were applied over and over again. Those rules produced a result tree that matched its own pattern. In some cases, the tree would grow in size each time the transformation was applied, in other cases, it stayed unmodified. In all cases, the formalization was clearly in error. In some cases, also the informal language definition needed to be changed, if it did not deal properly with recursive application of transformation rules.

[14] Some transformations were skipped erroneously, as their patterns did not match (even though they should have). Those errors were typically detected in a later phase of transformations, as the input was then not in the form that those later transformations expected. Various strategies to correct these problems had to be applied. For example, if transformations relied on the output of later transformations as a precondition for their own input, the order of transformations had to be changed.

Running this program eventually produces an execution trace, which then can be studied to analyze the formal meaning of the SDL specification originally taken as input. For example, when taking the SDL specification

```
use foo;
system sys:<<package foo>>syst;

package foo;
 system type syst;
  signal a, b;
  gate g in with a  out with b;
  state type stateT;
    start; nextstate s;
    state s; input a; output b; nextstate s; endstate;
  endsubstructure state type;
  state aggregation main_state: stateT;
 endsystem type;
endpackage;
```

as input, the resulting trace would read

```
*** Run of Specification generated by SDL Formal Compiler
** creation of agents, links, gates and states
*** environment sends signal 'a' to inDir Gate g
** agent for sys enters {'main_state'}
** agent for sys enters start transition of 'main_state'
** agent for sys enters {'s'}
** agent for sys receives signal 'a' in state 's'
** agent for sys sends signal 'b'
*** environment receives signal 'b' from outDir Gate g
*** Finishing Run of Specification
```

9 Correcting Errors

Each time the tools showed unexpected results, an analysis was necessary as to what precisely the problem was. In some cases, we found that simply the expectations were wrong, and that the observed behaviour is really a possible interpretation of the SDL system. In other cases, we traced the error to a bug in the tool chain. If the problem turned out to be an error in the formal semantics, this error was corrected. For this latter category, we counted all errors we corrected, and classified them further. The result of this classification is shown in table 1 below.

It should be noted that the large amount of errors is due to the fact that all errors encountered were counted, even if it was just one missing bracket or a missing parameter. Most of these errors were repaired with little effort.

In [15] it is shown how the tool chain can be used to check and improve the formal semantics definition and how to detect errors in the formal semantics.

10 Engineering Other Formal Language Definitions

While we developed the formal semantics and tool support only of a single programming language, SDL-2000, we believe that our approach is directly applicable to other programming languages. In order to apply our techniques, the four aspects of a formal semantics (lexis, syntax, static and dynamic semantics) need to be identified. It might be possible to extend the techniques to cover other aspects of language semantics (like non-functional aspects), but we have not studied such extensions.

Table 1. Errors found in the formal semantics

Category	Errors in F.2	Errors in F.3
Syntax errors	352	189
Type errors	88	314
Spelling Errors	115	80
Missing or superfluous parameters	159	34
Minor semantic errors	57	227
Incorrect usage of abstract syntax	144	83
New auxiliary functions		32
Problems with the abstract syntax 0	4	
Major semantic errors	12	3

For lexis and syntax, the major requirement is that the language has a textual notation, and that EBNF can be used to define a super-language (i.e. a grammar that produces a superset of all well-formed programs). Care should be taken to find a “small” super-language, as all programs which are syntactically correct but still ill-formed need to be detected by a well-formedness condition.

Many languages with a textual notation include short-hand notations. For example, in Java, not specifying a base class is equivalent to giving java.lang.Object as the base class. Defining transformation rules is a powerful technique for capturing the semantics of shorthand notations, and should be used wherever possible.

In our experience, using the same grammar on several abstraction levels (i.e. to represent programs with and without shorthands) causes no problems. Well-formedness conditions are best applied to the more abstract program representation, as less special cases have to be considered. “Early” checking of well-formedness conditions should only be applied were mandated by the informal language definition or to safeguard transformation rules.

To express the dynamic semantics, Abstract State Machines have been demonstrated to support various programming language features. However, in the compilation-based approach that we have used, it is more concise to transform the program into primitives of an abstract machine, and to use ASM programs then to give a meaning to such primitives.

Regardless of what formalisation techniques are used, it is essential to accompany them with tools that validate and execute the formal language definition. If a formal language definition uses the same notations as the SDL-2000 formal definition, the tools that we have developed can likely be used with little or no modification.

11 Comparison with Other Approaches

For the comparison of the SDL formal semantics with other formal semantics definitions, see e.g. [6]. In the scope of this paper, we only want to have a look at the executable formal semantics works and compiler construction tools.

It is not a completely new idea to define a formal semantics for a language. There were several other attempts to define programming language semantics formally. However, the direction for the formal semantics was different most of the time. The formal semantics started from the abstract grammar, and was hence used to prove certain properties of the language (good ones and bad ones). It was not really possible to derive a real compiler from the formal language description, because the front-end properties of the language, i.e. the compiler-related part, was not formalised. In addition, many of these attempts to formalise a language only targeted a subset of the language. For example, attempts to provide a formal semantics for the C and C++ programming languages ([13], [14]) ignore (among other things) the semantics of the pre-processor.

There have also been lots of attempts to define tools that generate compilers including the semantics of the languages, e.g. kimwitu [11], Eli (Paderborn) [12] and other compiler generation tools. It is possible to generate a compiler this way, however, the semantics of the formal texts used to derive the semantics of the language is not really given in a strict mathematical sense.

12 Summary and Conclusions

Defining a formal semantics for a real-world language (such as SDL-2000) is a tedious and error-prone work. The advantages of developing such a formal definition (avoidance of ambiguity in the language definition, ability to proof properties of systems) can only materialise if the formal definition itself is nearly free of errors, and procedures are in place to correct errors that have been found.

In our experience, designing and formulating a formal language definition are only a small part of making it actually useful. It needs to be accompanied with a tool chain that helps to validate the formal definition in itself, and gives practitioners of the language a tool to access the formal definition more conveniently.

Acknowledgements

We would like to thank the SDL Forum Society and Microsoft Research for funding this research, and providing access to both the SDL standard, and beta versions of the AsmL compiler.

References

1. A. Prinz. *Formal Semantics for SDL.* Definition and Implementation. Humboldt-Universität zu Berlin, 2001.
2. International Telecommunication Union (ITU). *Specification and Description Language (SDL).* ITU-T Recommendation Z.100. Geneva, 2000.
3. International Telecommunication Union (ITU). *SDL Formal Definition.* ITU-T Recommendation Z.100.F, Geneva, 2000.
4. Michael Piefel. *Kimwitu++*. http://site.informatik.hu-berlin.de/kimwitu++.

5. Y. Gurevich. Evolving Algebras 1993: *Lipari Guide*. In E. Börger, editor, *Specification and Validation Methods*, pages 9-36, Oxford University Press, 1995
6. Eschbach, R., Glässer, U., Gotzhein, R., von Löwis, M., Prinz, A.: *Formal Definition of SDL-2000*: Compiling and Running SDL Specifications as ASM Models, Journal of Universal Computer Science 7 (11), 2001, Springer, pp. 1025-1050
7. Microsoft Research. *AsmL*. http://research.microsoft.com/foundations/AsmL/default.html
8. M. v. Löwis, M. Piefel. The Term Processor Kimwitu++. In N. Callaos, T. Leng, B. Sanchez: Proceedings of the 6th World Conference on Systemics, Cybernetics, and Informatics, Orlando, 2002
9. Microsoft Corporation. Microsoft® Office 2000 Resource Kit. Microsoft Press, 1999
10. T. Archer. Inside C#. Microsoft Press, 2001
11. P. van Eijk. Tools for LOTOS, a Lotosphere overview. In tutorial proceedings of 11th Symposium on Protocol Specification, Testing and Verification, Sydney, 1991.
12. U. Kastens. Executable Specifications for Language Implementation. In Fifth International Symposium on Programming Language Implementations and Logic Programming, Tallinn, 1993, Springer LNCS 714, pp. 1-11.
13. Y. Gurevich, J. K. Huggins. The Semantics of the C Programming Language. Springer LNCS 702, Springer, 1993.
14. C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, Specification and Validation Methods, Oxford University Press, 1995.
15. A. Prinz, M. v. Löwis: Generating A Compiler for SDL From The Formal Language Definition in R. Reed (Ed.): SDL 2003: System Design, Proceedings of the 11th International SDL Forum 2003, LNCS 2708, Springer, 2003.

A Syntax-Directed Hoare Logic for Object-Oriented Programming Concepts

Cees Pierik[1] and Frank S. de Boer[1,2]

[1] Institute of Information and Computing Sciences
Utrecht University, The Netherlands
[2] CWI, Amsterdam, The Netherlands
{cees,frankb}@cs.uu.nl

Abstract. This paper outlines a sound and complete Hoare logic for a sequential object-oriented language with inheritance and subtyping like Java. It describes a weakest precondition calculus for assignments and object-creation, as well as Hoare rules for reasoning about (mutually recursive) method invocations with dynamic binding. Our approach enables reasoning at an abstraction level that coincides with the general abstraction level of object-oriented languages.

1 Introduction

The concepts of inheritance and subtyping in object-oriented programming have many virtues. But they also pose challenges for reasoning about programs. For example, subtyping enables variables with different types to be aliases of the same object, and it destroys the static connection between a method call and its implementation. Inheritance, without further restrictions, adds complexity by permitting objects to have different instance variables with the same identifier.

This paper outlines a Hoare logic for a sequential object-oriented language that contains all standard object-oriented features, including inheritance, subtyping and dynamic binding. The logic consists of a weakest precondition calculus for assignments and object-creation, as well as Hoare rules for reasoning about (mutually recursive) method invocations with dynamic binding. The resulting logic is complete in the sense that any valid correctness formula can be derived within the logic.

The Hoare logic presented in this paper is *syntax-directed*. By a syntax-directed Hoare logic we mean a Hoare logic that is based on an assertion language of the same abstraction level as the programming language. In particular, there is no explicit reference to the object store in our assertion language, as opposed to [1]. Moreover, our Hoare logic is based on a weakest precondition calculus that consists of purely *syntactical* substitution operations.

Hoare introduced the axiom $\{P[e/x]\}\ x := e\ \{P\}$ for reasoning about simple assignments in his seminal paper [2]. A *semantical* variant would be

$$\{P[\sigma\{x := e\}/\sigma]\}\ x := e\ \{P\}\ ,$$

Table 1. The syntax of cOORe.

Below, the operator `op` is an arbitrary operator on elements of a primitive type, and m is an arbitrary identifier.

$e \in \text{Expr}$	$::= \texttt{null} \mid \texttt{this} \mid u \mid e.x \mid (C)e \mid e \ \texttt{instanceof}\ C \mid \texttt{op}(e_1, \ldots, e_n)$
$y \in \text{Loc}$	$::= u \mid e.x$
$s \in \text{SExpr}$	$::= \texttt{new}\ C() \mid u.m(e_1, \ldots, e_n) \mid \texttt{super}.m(e_1, \ldots, e_n)$
$S \in \text{Stat}$	$::= y = e\ ; \mid y = s\ ; \mid S\ S \mid \texttt{if}\ (e)\ \{\ S\ \}\ \texttt{else}\ \{\ S\} \mid \texttt{while}\ (e)\ \{\ S\ \}$
$meth \in \text{Meth}$	$::= m(u_1, \ldots, u_n)\ \{\ S\ \texttt{return}\ e\ ;\ \}$
$main \in \text{Main}$	$::= \texttt{main}()\ \{\ S\ \}$
$exts \in \text{Exts}$	$::= \epsilon \mid \texttt{extends}\ C$
$class \in \text{Class}$	$::= \texttt{class}\ C\ exts\ \{\ meth^*\ \}$
$\pi \in \text{Prog}$	$::= class^*\ main$

where $\sigma\{x := e\}$ denotes the state that results from σ by assigning $\sigma(e)$ to x. Here, the occurrence of σ shows the employed representation of the state, and state updates like $\sigma\{x := e\}$ reveal the encoding of the semantics. In the original approach, assertions have the same abstraction level as the programming language and hide all these details.

Another advantage of the syntax-directed approach can be explained by the following example. Suppose we want to prove $\{y = 1\}\ x := 0\ \{y = 1\}$. Using our approach, this amounts to proving the implication $y = 1 \rightarrow y = 1$. The semantical approach requires proving $\sigma(y) = 1 \rightarrow \sigma\{x := 0\}(y) = 1$. A theorem prover must do one additional reasoning step in this case, namely resolving that y is a different location than x. This step is otherwise encoded in the substitution. The minor difference in this example leads to larger differences, for example when reasoning about aliases. Our substitution operation precisely reveals in which cases we have to check for possible aliases. Finally, observe that the semantical approach requires an encoding of (elements of) the semantics of the programming language in the theorem prover.

This paper is organized as follows. In Sect. 2 and 3 we introduce the programming language and the assertion language. Section 4 and 5 describe the weakest precondition calculus for assignments and object creation. In Sect. 6 we give Hoare rules for reasoning about method calls. Related research is discussed in the last section. The completeness proof and other details of the logic are described in the full version of this paper [3].

2 The Object-Oriented Language

The language we consider in this paper (dubbed cOORe) contains all standard object-oriented features like inheritance and subtyping. For ease of reading, we adopted the syntax of the corresponding subset of Java. The syntax of cOORe can be found in Table 1.

The primitive types we consider are `boolean` and `int`. We assume given a set $\mathcal{C}$ of *class names*, with typical element C. The set of types $\mathcal{T}$ is the union of the set $\{\texttt{int}, \texttt{boolean}\}$ and $\mathcal{C}$. In the sequel, t will be a typical element of $\mathcal{T}$. The language is strongly-typed. By $[\![e]\!]$ we denote the (declared) static type of expression e. The type of `this` is determined by its context. We will silently assume that all expressions are well-typed.

By TVar we denote the set of local (or temporary) variables. Each class C is equipped with a set of instance variables IVar_C. We use u and x as typical elements of the sets TVar and IVar_C, respectively. The location y is either a local variable or an instance variable. Instance variables belong to a specific object and store its internal state. Local variables belong to a method and last as long as this method is active. A method's formal parameters are also local variables.

A program in cOOre is a finite set of classes and a main method which initiates the execution of the program. A class defines a finite set of methods. A method m consists of its formal parameters $u_1, \ldots, u_n$, a statement S, and an expression e without side effect which denotes the return value. A clause `class` C `extends` C' indicates that class C is a subclass of C'. It implies that class C inherits all methods and instance variables of class C'.

The expression $e.x$ refers to the instance variable x of object e as found in class $[\![e]\!]$ or, if not present in $\mathrm{IVar}_{[\![e]\!]}$, the first occurrence of this variable in a superclass of $[\![e]\!]$. Observe that a class C can hide an inherited instance variable x by defining another instance variable x. An expression $e.x$, with $[\![e]\!] = C$, will then refer to the new variable. The cast operator (C) in $(C)e$ changes the type of expression e to C. Thus it can be used to access hidden variables. For example, $((C)\mathtt{this}).x$ denotes the first occurrence of an instance variable x of object `this` as found by an upward search starting in class C. This might be a variable different from $\mathtt{this}.x$. We assume that $[\![e]\!]$ in $(C)e$ is a reference type. An expression e `instanceof` C is `true` if e is non-null and refers to an instance of (a subclass of) class C.

We have the usual assignments of expressions to variables. An assignment $y = \mathtt{new}\ C()$ involves the creation of an object of class C. Note that the language does not include constructor methods declarations. Thus an expression like $\mathtt{new}\ C()$ will call the default constructor method, which will assign to all instance variables their default value.

Observe that the callee of a method call can be denoted by a local variable only. We assume that all methods are public. An assignment $y = u.m(e_1, \ldots, e_n)$ involves a call of method m of the object denoted by the local variable u. These calls are bound dynamically to an implementation, depending on the class of the object denoted by u. Calls of the form $y = \mathtt{super}.m(e_1, \ldots, e_n)$ are bound statically. The corresponding implementation is found by searching upwards in the class hierarchy for a definition of m, starting in the superclass of $[\![\mathtt{this}]\!]$.

The language cOOre permits only side effects in the outermost operator. This is a common restriction in Hoare logics that clarifies the presentation of the logic. However, it is not essential. Early work by Kowaltowski already introduces a general approach to side effects [4]. On the other hand, one could argue that the restriction on side effects leads to more reliable programs. Gosling et al. remark: 'Code is usually clearer when each expression contains at most one side effect, as its outermost operation, and when code does not depend on exactly which exception arises as a consequence of the left-to-right evaluation of expressions.' [5, p. 305]

2.1 Semantics

In this section, we only describe the semantics of expressions because this suffices to understand the rest of the paper. The semantics of cOORe is defined in terms of a representation of the state of an object-oriented program and a subtype relation.

By $t \preceq t'$ we denote that t is a subtype of t'. The relation $\preceq$ is given by the class definitions in the program. The declaration `class A extends B` implies that $A \lhd B$ (where $A \lhd B$ denotes that class A is a direct subclass of class B). In fact, the $\lhd$ relation is a partial function that defines the superclass of a class. Therefore we will assume that $F_{\lhd}(C)$ denotes the direct superclass of a class C. The partial order $\preceq$ is the reflexive, transitive closure of the $\lhd$ relation. We say that t' is a *proper* subtype of t, denoted by $t' \prec t$, if $t' \preceq t$ and $t' \neq t$.

We represent objects as follows. Each object has its own identity and belongs to a certain class. For each class $C \in \mathcal{C}$ we introduce therefore the infinite set $O^C = \{C\} \times \mathbb{N}$ of object identities in class C (here $\mathbb{N}$ denotes the set of natural numbers). Let $\mathtt{subs}(C)$ be the set $\{C' \in \mathcal{C} | C' \preceq C\}$. By $\mathrm{dom}(C)$ we denote the set $(\bigcup_{C' \in \mathtt{subs}(C)} O^{C'}) \cup \{\bot\}$. Here $\bot$ is the value of `null`. In general, $\bot$ stands for 'undefined'. For $t = \mathtt{int}, \mathtt{boolean}$, $\mathrm{dom}(t)$ denotes the set of boolean and integer values, respectively.

The internal state of an object $o \in O^C$ is a total function that maps the instance variables of class C and its superclasses to their values. Let $\mathtt{supers}(C)$ be the set $\{C' \in \mathcal{C} | C \preceq C'\}$. The internal state of an instance of class C is an element of the set internal(C), which is defined by the (generalized) cartesian product

$$\prod_{C' \in \mathtt{supers}(C)} \Big(\prod_{x \in \mathrm{IVar}_{C'}} \mathrm{dom}([\![x]\!]) \Big) \ .$$

A configuration σ is a partial function that maps each *existing* object to its internal state. We will assume that σ is an element of the set Σ, where

$$\Sigma = \prod_{C \in \mathcal{C}} \Big(\mathbb{N} \rightharpoonup \mathrm{internal}(C) \Big) \ .$$

In the sequel, we will write $\sigma(o)$ for some object $o = (C, n)$ as shorthand for $\sigma(C)(n)$. In this way, $\sigma(o)$ denotes the internal state of an object. It is not defined for objects that do not exist in a particular configuration σ. Thus σ specifies the set of existing objects. We will only consider configurations that are *consistent*. We say that a configuration is consistent if no instance variable of an existing object refers to a non-existing object.

The local context $\tau \in \mathrm{T}$ specifies the active object and the values of the local variables. Formally, T is the set

$$(\bigcup_{C \in \mathcal{C}} \mathrm{dom}(C)) \times \prod_{u \in \mathrm{TVar}} \mathrm{dom}([\![u]\!]).$$

The first component of any τ is the active object and the second component is a function which assigns to every local variable u its value. The first component

Table 2. Evaluation of expressions.

$$
\begin{array}{rcl}
\mathcal{E}(\texttt{null})(\sigma,\tau) & = & \bot \\
\mathcal{E}(\texttt{this})(\sigma,\tau) & = & \tau(\texttt{this}) \\
\mathcal{E}(u)(\sigma,\tau) & = & \tau(u) \\
\mathcal{E}(e.x)(\sigma,\tau) & = & \begin{cases} \bot & \text{if } \mathcal{E}(e)(\sigma,\tau)=\bot \\ \sigma(\mathcal{E}(e)(\sigma,\tau))(\mathsf{resolve}(\llbracket e \rrbracket)(x))(x) & \text{otherwise} \end{cases} \\
\mathcal{E}(\ (C)e\ )(\sigma,\tau) & = & \begin{cases} \bot & \text{if } \mathcal{E}(e)(\sigma,\tau)=(C',n) \text{ and } C' \not\preceq C \\ \mathcal{E}(e)(\sigma,\tau) & \text{otherwise} \end{cases} \\
\mathcal{E}(e \ \texttt{instanceof}\ C)(\sigma,\tau) & = & \begin{cases} \mathsf{false} & \text{if } \mathcal{E}(e)(\sigma,\tau)=\bot \\ C' \preceq C & \text{if } \mathcal{E}(e)(\sigma,\tau)=(C',n) \end{cases} \\
\mathcal{E}(\mathsf{op}(e_1,\ldots,e_n))(\sigma,\tau) & = & \begin{cases} \bot & \text{if } e'_i=\bot \text{ for some } i=1,\ldots,n \\ \bar{\mathsf{op}}(e'_1,\ldots,e'_n) & \text{otherwise,} \end{cases} \\
 & & \text{where } \bar{\mathsf{op}} \text{ denotes the fixed interpretation of } \mathsf{op}\text{, and} \\
 & & e'_i = \mathcal{E}(e_i)(\sigma,\tau)\text{, for } i=1,\ldots,n.
\end{array}
$$

will be $\bot$ if there is no active object, which is the case during execution of the main method. In the sequel, we denote the first component o of a local context $\tau = \langle o, f\rangle$ by $\tau(\texttt{this})$ and $f(u)$ by $\tau(u)$. Although the local state of the main method can be $\langle \bot, f\rangle$, we will assume in other methods that the first element is an existing object. A local state is consistent with a global configuration if all local variables do not refer to non-existing objects. A state is a pair (σ,τ), where the local context τ is required to be consistent with the configuration σ.

To find fields in an internal state we need a way to determine in which class a field is declared. As explained above, the type of the quantifier e determines to which field an expression $e.x$ refers. We introduce a function $\mathsf{resolve}$ that yields the class of the field to which the expression $e.x$ refers given $\llbracket e \rrbracket$ and x. It is defined as follows.

$$
\mathsf{resolve}(C)(x) = \begin{cases} C & \text{if } x \in \mathrm{IVar}_C \\ \mathsf{resolve}(F_{\lhd}(C))(x) & \text{otherwise} \end{cases}
$$

Expressions are evaluated relative to a subclass relation $\preceq$, a configuration σ, and a local context τ. The result of the evaluation of an expression e is denoted by $\mathcal{E}(e)(\sigma,\tau)$. The $\preceq$ relation is left implicit. The definition of $\mathcal{E}$ can be found in Table 2.

3 The Assertion Language

The proof system is tailored to a specific assertion language called AsO (Assertion language for Object structures). The syntax of AsO is defined by the following grammar.

$$
\begin{array}{rl}
l \in \mathrm{LExpr} ::= & \texttt{null} \mid \texttt{this} \mid u \mid z \mid l.x \mid (C)l \mid l_1 = l_2 \mid l \ \texttt{instanceof}\ C \\
 & \mid \mathsf{op}(l_1,\ldots,l_n) \mid \texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi} \\
P, Q \in \mathrm{Ass} ::= & l_1 = l_2 \mid \neg P \mid P \wedge Q \mid \exists z : t(P)
\end{array}
$$

Table 3. Evaluation of assertions.

$$\begin{aligned}
\mathcal{L}(z)(\sigma,\tau,\omega) &= \omega(z) \\
\mathcal{L}(l_1 = l_2)(\sigma,\tau,\omega) &= \begin{cases} \mathsf{true} \text{ if } \mathcal{L}(l_1)(\sigma,\tau,\omega) = \mathcal{L}(l_2)(\sigma,\tau,\omega) \\ \mathsf{false} \text{ otherwise} \end{cases} \\
\mathcal{L}(\texttt{if } l_1 \texttt{ then } l_2 \texttt{ else } l_3 \texttt{ fi})(\sigma,\tau,\omega) &= \begin{cases} \bot & \text{if } \mathcal{L}(l_l)(\sigma,\tau,\omega) = \bot \\ \mathcal{L}(l_2)(\sigma,\tau,\omega) & \text{if } \mathcal{L}(l_1)(\sigma,\tau,\omega) = \mathsf{true} \\ \mathcal{L}(l_3)(\sigma,\tau,\omega) & \text{if } \mathcal{L}(l_1)(\sigma,\tau,\omega) = \mathsf{false} \end{cases}
\end{aligned}$$

In the assertion language we assume a set of (typed) *logical* variables LVar with typical element z. We include expressions of the form $(C)l$ to be able to access hidden instance variables. The use of l `instanceof` C will become clear in Sect. 6. We sometimes omit the type information in $\exists z : t(P)$ if it is clear from the context.

The assertion language is strongly-typed similar to the programming language. Logical variables can additionally have type t^*, for some t in the old set of types $\mathcal{T}$. This means that its value is a finite sequence of elements of $\mathrm{dom}(t)$. Therefore $\mathrm{dom}(t^*)$ is the set of finite sequences of elements of $\mathrm{dom}(t)$.

Assertion languages for object-oriented programs inevitably contain expressions like $l.x$ and $(C)l$ that are normally undefined if, for example, l is `null`. However, as an assertion their value should be defined. We solved this problem by giving such expression the same value as `null`. By only allowing the non-strict equality operator as a basic formula, we nevertheless ensure that formulas are two-valued. If we omit this operator the value is implicitly compared to true. An alternative solution which is employed in JML [6] is to return an arbitrary element of the underlying domain.

Logical expressions are evaluated relative to a subclass relation $\preceq$, a configuration σ, a local context τ, and a logical environment $\omega \in \prod_{z\in\mathrm{LVar}} \mathrm{dom}([\![z]\!])$, which assigns values to the logical variables. The logical environment is restricted similar to a local context: no logical variable is allowed to point to an object that does not exist in the current configuration.

The result of the evaluation of an expression l is denoted by $\mathcal{L}(l)(\sigma,\tau,\omega)$. Again, we leave the $\preceq$ relation implicit. The function $\mathcal{L}$ is similar to $\mathcal{E}$ for all constructs that are present in cOORe. All new cases are listed in Table 3.

A formula $\exists z : C(P)$ states that P holds for an *existing* instance of (a subclass of) C or `null`. Thus the quantification domain of a variable depends not only on the type of the variable but also on the configuration. Let $\mathrm{qdom}(t,\sigma)$ denote the quantification domain of a variable of type t in configuration σ. We define $\mathrm{qdom}(C,\sigma) = \{o \in \mathrm{dom}(C) | \sigma(o) \text{ is defined }\} \cup \{\bot\}$. A formula $\exists z : C^*(P)$ states the existence of a sequence of existing objects. Therefore, we define

$$\mathrm{qdom}(C^*,\sigma) = \{\alpha \in \mathrm{dom}(C^*) | \forall n \in \mathbb{N}.\alpha[n] \in \mathrm{qdom}(C,\sigma)\} \ .$$

Finally, we have $\mathrm{qdom}(t,\sigma) = \mathrm{dom}(t)$ for $t \in \{\texttt{int}, \texttt{boolean}, \texttt{int}^*, \texttt{boolean}^*\}$.

The evaluation of a formula P can be defined similar to the evaluation of a logical expression. The resulting value is denoted by $\mathcal{A}(P)(\sigma,\tau,\omega)$. The only interesting case is $\mathcal{A}(\exists z : t(P))(\sigma, \tau, \omega)$, which yields true if $\mathcal{A}(P)(\sigma, \tau, \omega\{\alpha/z\}) = \mathsf{true}$ for some $\alpha \in \text{qdom}(t, \sigma)$ and false otherwise.

The standard abbreviations like $\forall z P$ for $\neg\exists z \neg P$ are valid. The statement $\sigma, \tau, \omega \models P$ means that $\mathcal{A}(\mathrm{P})(\sigma, \tau, \omega)$ yields true.

4 Assignments and Aliasing

This section shows how we model simple assignments by means of a substitution operation. The basic underlying idea as originally introduced in [2] is that the assertion that results from the substitution has the same meaning in the state before the assignment as the unmodified assertion in the state after the assignment. In other words, the substitution computes the *weakest precondition.*

First we observe that given an assignment $u = e$, and a *postcondition* P, the assertion $P[e/u]$ obtained from P by replacing every occurrence of u by e is *not* the weakest precondition. Subtyping combined with dereferencing is the cause of this phenomenon. Subtyping allows u and e to have different types. This implies that the substitution $[e/u]$ might change the type of an expression l. The only restriction imposed by the language is that $[\![e]\!] \preceq [\![u]\!]$.

To see where things go wrong, consider the following case. Suppose we have two classes C_1 and C_2 such that $C_2 \prec C_1$. Furthermore, assume that in each of the two classes an instance variable x of type `int` is defined. Finally, suppose that we have two local variables u_1 and u_2, such that $[\![u_1]\!] = C_1$ and $[\![u_2]\!] = C_2$. Now consider the specification of the following assignment.

$$\{u_2.x = 3\}\ u_1 = u_2;\ \{u_1.x = 3\}$$

This specification is not valid. Clearly, we have that $u_1.x = 3[u_2/u_1] \equiv u_2.x = 3$ (denoting syntactical equality by $\equiv$). But the expressions $u_1.x$ and $u_2.x$ point to different locations, even if u_1 and u_2 refer to the same object, because the types of u_1 and u_2 are different. A correct specification would be

$$\{((C_1)u_2).x = 3\}\ u_1 = u_2;\ \{u_1.x = 3\}\ .$$

The above specification presents the key to the solution of this problem. The result of the substitution $[e/u]$ should be changed in such a way that the types remain unchanged. This can be done by changing the result of $u[e/u]$ to $\mathsf{cast?}([\![u]\!], e)$. The auxiliary function $\mathsf{cast?}(t, l)$ is defined as follows.

$$\mathsf{cast?}(t, l) = \begin{cases} l & \text{if } t \text{ is a primitive type} \\ (t)l & \text{otherwise} \end{cases}$$

All other cases of the substitution $[e/u]$ correspond to the standard notion of (structural) substitution. We will assume in the rest of this paper that a substitution of the form $[e/u]$ corresponds to this modified substitution operation. The following theorem states that $P[e/u]$ is the weakest precondition of P with respect to the assignment $u = e$.

Theorem 1. *If* $[\![e]\!] \preceq [\![u]\!]$ *we have*

$$\sigma, \tau, \omega \models P[e/u] \text{ if and only if } \sigma, \tau', \omega \models P \enspace ,$$

where τ' *denotes the local state that results from* τ *by assigning* $\mathcal{E}(e)(\sigma, \tau)$ *to* u.

The above theorem justifies the axiom $\{P[e/u]\}\ u = e\ \{P\}$. A crucial part of the proof consists of showing type preservation of $[e/u]$, that is, proving that $[\![l[e/u]]\!] = [\![l]\!]$.

For the same reason, the usual notion of substitution does not suffice for an assignment $e.x = e'$. But such assignments are also complicated because of possible aliases of the location $e.x$, namely expressions of the form $l.x$. It is possible that l refers to the object denoted by e (before the assignment), which implies that $l.x$ denotes the same location as $e.x$ and should be substituted by e'. If l does not refer to object e no substitution should take place. If we cannot decide between these possibilities by the form of the expression and their types, a conditional expression is constructed which decides dynamically.

The definition of the substitution operation $[e'/e.x]$ is straightforward in most cases. The most interesting case is $l.x[e'/e.x]$. It results in one of the two following expressions. If $[\![l]\!] \preceq [\![e]\!]$ or $[\![e]\!] \preceq [\![l]\!]$ and, moreover, $\mathsf{resolve}([\![l]\!])(x) = \mathsf{resolve}([\![e]\!])(x)$, we have

$$l.x[e'/e.x] \equiv \texttt{if}\ l[e'/e.x] = e\ \texttt{then}\ \mathsf{cast?}([\![l.x]\!], e')\ \texttt{else}\ (l[e'/e.x]).x\ \texttt{fi} \enspace .$$

Otherwise, we simply have $l.x[e'/e.x] \equiv (l[e'/e.x]).x$. This can be explained as follows. Note that if $[\![l]\!] \not\preceq [\![e]\!]$ and $[\![e]\!] \not\preceq [\![l]\!]$ then the expressions l and e cannot refer to the same object, because the domains of $[\![l]\!]$ and $[\![e]\!]$ are disjoint. The clause $\mathsf{resolve}([\![l]\!])(x) = \mathsf{resolve}([\![e]\!])(x)$ checks if the two occurrences of x denote the same instance variable. We define $l.y[e'/e.x] \equiv (l[e'/e.x]).y$ if x and y are syntactically different. The definition is extended to assertions in the standard way.

As a simple example, we consider the assignment $\texttt{this}.x = 0$ and the postcondition $u.y.x = 1$, where x and y are instance variables and u is a local variable. Considering types, we assume in this example that $[\![u.y]\!] = [\![\texttt{this}]\!] = C$ and $[\![u]\!] = C'$. Applying the corresponding substitution $[0/\texttt{this}.x]$ to the assertion $u.y.x = 1$ results in the assertion

$$(\texttt{if}\ u.y = \texttt{this}\ \texttt{then}\ 0\ \texttt{else}\ u.y.x\ \texttt{fi}) = 1 \enspace .$$

This assertion clearly is logically equivalent to $\neg(u.y = \texttt{this}) \wedge u.y.x = 1$.

The following theorem states that $P[e'/e.x]$ is indeed the weakest precondition of the assertion P with respect to the assignment $e.x = e'$. Thus it justifies the axiom $\{P[e'/e.x]\}e.x = e'\{P\}$. Its proof again requires showing type preservation of the substitution.

Theorem 2. *If* $[\![e']\!] \preceq [\![e.x]\!]$ *and* $\mathcal{E}(e)(\sigma, \tau) \neq \bot$ *we have*

$$\sigma, \tau, \omega \models P[e'/e.x] \text{ if and only if } \sigma', \tau, \omega \models P \enspace ,$$

where $\sigma'(o)(\mathsf{resolve}([\![e]\!])(x))(x) = \mathcal{E}(e')(\sigma, \tau)$, *for* $o = \mathcal{E}(e)(\sigma, \tau)$, *and in all other cases* σ *agrees with* σ'.

5 Object Creation

Next we consider the creation of objects. Our goal is to define a substitution $[\texttt{new}\ C/u]$ which computes the weakest precondition of the assignment $u = \texttt{new}\ C()$. Note that an assignment $e.x = \texttt{new}\ C()$ can be simulated by the sequence of assignments $u = \texttt{new}\ C(); e.x = u$, where u is a fresh local variable. The weakest precondition of an assignment $e.x = \texttt{new}\ C$ w.r.t. postcondition P is therefore $P[u/e.x][\texttt{new}\ C/u]$, where u is a fresh local variable which does not occur in P and e.

For certain logical *expressions* l we cannot define a weakest precondition $l[\texttt{new}\ C/u]$, because they refer to the new object, and there is no expression that refers to this object in the state before its creation, because it does not exist in that state. Therefore the result of the substitution must be left undefined in some cases. However, we can define the substitution on any logical expression l that is not of the form u, $(C')u$ or $\texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi}$ with $[\![l_2]\!] = [\![l_3]\!] \in \mathcal{C}$. This suffices to define $[\texttt{new}\ C/u]$ on any *formula*. We will do so by means of a contextual analysis of the occurrences of u.

To simplify the definition of $[\texttt{new}\ C/u]$ we start by rewriting logical expressions into a normal form. This proceeds in two steps. Firstly, we remove all occurrences of casts of conditional expressions by means of the following equivalence.

$$(C)\texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi} = \texttt{if}\ l_1\ \texttt{then}\ (C)l_2\ \texttt{else}\ (C)l_3\ \texttt{fi}$$

Secondly, we remove all expressions of the form $(C')(C'')u$. Observe that such an expression is either equivalent to $(C')u$ or $(C')\texttt{null}$, depending on the validity of $C \preceq C''$. In both steps we replace an expression by an expression of the same type.

Due to space limitations we cannot give all cases of $l[\texttt{new}\ C/u]$. But we trust that the general idea becomes clear by considering the following examples, starting with $l.x[\texttt{new}\ C/u]$. In general, we have to give special attention to cases where $l[\texttt{new}\ C/u]$ may be undefined. Therefore we single out the case where $l \equiv u$.

$$u.x[\texttt{new}\ C/u] \equiv \begin{cases} \mathsf{false} & \text{if } [\![u.x]\!] = \texttt{boolean} \\ 0 & \text{if } [\![u.x]\!] = \texttt{int} \\ \texttt{null} & \text{otherwise} \end{cases}$$

This example shows that we can find an equivalent expression even if the substitution is undefined for l. The instance variables of a new object have their default values after creation. These values depend on the types of the variables, which is reflected by the above substitution. The case where $l \equiv (C')u$ is similar, but requires additionally checking if $C \preceq C'$ to predict if the cast will succeed. If the cast fails, one can return `null`.

The other special case is that of a conditional expression. Suppose that $[\![(\texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi}).x]\!] = C'$. Then we define

$$\begin{aligned}&(\texttt{if}\ l_1\ \texttt{then}\ l_2\ \texttt{else}\ l_3\ \texttt{fi}).x[\texttt{new}\ C/u] \\ &\equiv \texttt{if}\ l_1[\texttt{new}\ C/u]\ \texttt{then}\ ((C')l_2).x[\texttt{new}\ C/u]\ \texttt{else}\ ((C')l_3).x[\texttt{new}\ C/u]\ \texttt{fi}\ .\end{aligned}$$

In all other cases we have $l.x[\texttt{new}\ C/u] \equiv (l[\texttt{new}\ C/u]).x$.

The changing scope of a bound occurrence of a variable z ranging over objects, as induced by the creation of a new object, is captured as follows. We define $(\exists z : C'(P))[\texttt{new}\ C/u]$

$$\equiv \begin{cases} (\exists z : C'(P[\texttt{new}\ C/u])) \vee (P[(C')u/z][\texttt{new}\ C/u]) & \text{if } C \preceq C' \\ \exists z : C'(P[\texttt{new}\ C/u]) & \text{otherwise.} \end{cases}$$

The idea of the application of $[\texttt{new}\ C/u]$ to $(\exists z P)$ is that the first disjunct $\exists z(P[\texttt{new}\ C/u])$ represents the case that P holds for an old object whereas the second disjunct $P[(C')u/z][\texttt{new}\ C/u]$ represents the case that the new object itself satisfies P. The substitution $[(C')u/z]$ consists of simply replacing every occurrence of z by $(C')u$. The other cases of $l[\texttt{new}\ C/u]$ and $P[\texttt{new}\ C/u]$ can be found in [3].

The following theorem states that $P[\texttt{new}\ C/u]$ indeed calculates the weakest precondition of P (with respect to the assignment $u = \texttt{new}\ C()$).

Theorem 3. *Let $[\![u]\!] \preceq C$. Then we have*

$$\sigma, \tau, \omega \models P[\texttt{new}\ C/u] \text{ if and only if } \sigma', \tau', \omega \models P \ ,$$

where σ' is obtained from σ by extending the domain of σ with a new object $o = (C, n) \notin \mathrm{qdom}(C, \sigma)$ and setting its instance variables at their default values. Furthermore, the resulting local context τ' is obtained from τ by assigning o to the variable u.

6 Method Invocations

In this section we discuss the rules for method invocations. In particular, we will analyze reasoning about dynamically bound method calls like in the statement $S \equiv y = u.m(e_1, \ldots, e_n)$. A correctness formula $\{P\}S\{Q\}$ implies that Q holds after the call independent of which implementation is executed. Therefore we must consider all implementations of m that are defined in (a subclass of) $[\![u]\!]$, and the implementation that is inherited by class $[\![u]\!]$ if it does not contain an implementation of method m itself.

The challenge in this section is to show that our assertion language is able to express the conditions under which an implementation is bound to a particular call given the restriction imposed by the abstraction level. That is, by using only expressions from the programming language. Secondly, we aim to define and present the rules in such a way that their translation to proof obligations in proof outlines for object-oriented programs is straightforward. For both these reasons we cannot adopt the virtual methods approach as proposed in [1].

We first consider a statement of the form $y = \texttt{super}.m(e_1, \ldots, e_n)$, because this allows us to explain many features of our approach while postponing the complexity of late binding. Suppose that the statement occurred somewhere in the definition of a class C. Assume that searching for the definition of m starting in the superclass of C ends in class C' with the following implementation $m(u_1, \ldots, u_n)\ \{\ S\ \texttt{return}\ e\ \}$. Then the invocation $\texttt{super}.m(e_1, \ldots, e_n)$

is bound to this particular implementation. The following rule for overridden method invocations (OMI) allows the derivation of a correctness specification for $y = \texttt{super}.m(e_1, \ldots, e_n)$ from a correctness specification of the body S of the implementation of m.

$$\frac{\{P' \wedge I\} S \{Q'[e/\texttt{return}]\} \quad Q'[(C')\texttt{this}/\texttt{this}][\bar{f}/\bar{z}] \rightarrow Q[\texttt{return}/y]}{\{P'[(C')\texttt{this}, \bar{e}/\texttt{this}, \bar{u}][\bar{f}/\bar{z}]\}\ y = \texttt{super}.m(e_1, \ldots, e_n)\ \{Q\}} \quad \text{(OMI)}$$

The precondition P' and postcondition Q' of S are transformed into corresponding conditions of the call by the substitution $[(C')\texttt{this}/\texttt{this}]$. This substitution reflects the context switch. The active object is the same in both contexts, but its type differs. The substitution corrects this. It corresponds to the standard notion of structural substitution, but should take place simultaneously with the (also simultaneous) substitutions $[\bar{e}/\bar{u}]$. These substitutions model the assignment of the actual parameters $\bar{e} = e_1, \ldots, e_n$ to the formal parameters $\bar{u} = u_1, \ldots, u_n$. Note that we have for every formal parameter u_i and corresponding actual parameter e_i that $[\![e_i]\!] \preceq [\![u_i]\!]$. So the simultaneous substitution we mean here is the generalization of $[e/u]$ as defined in Sect. 4. Except for the formal parameters $u_1, \ldots, u_n$, no other local variables are allowed in P'. We do not allow local variables in Q'.

We cannot simply substitute y by the result value e in Q, because e may refer to local variables of S that might clash with local variables of the caller. Therefore we introduce a logical variable `return`. The substitution $[e/\texttt{return}]$ applied to the postcondition Q' of S in the first premise models a (virtual) assignment of the result value to the logical variable `return`, which must not occur in the assertion Q. The related substitution $[\texttt{return}/y]$ applied to the postcondition Q of the call models the actual assignment of the return value to y. The substitution corresponds to one of the enhanced notions of substitution as defined in Sect. 4.

The assertion I in the precondition of S specifies the initial values of the local variables of m (excluding its formal parameters): In cOORE we have $u = \mathsf{false}$, in case of a boolean local variable, $u = 0$, in case of an integer variable, and $u = \texttt{null}$, for a reference variable.

Next we observe that the local state of the caller is not affected by the execution of S by the receiver. For this reason we know that an expression that only refers to local variables of the caller or the keyword `this` is constant during a method call. Such an expression f is generated by the following abstract syntax.

$$\begin{aligned} f ::= {} & \texttt{null} \mid \texttt{this} \mid u \mid (C)f \mid f_1 = f_n \mid f \texttt{ instanceof } C \mid \texttt{op}(f_1, \ldots, f_n) \\ & \mid \texttt{if } f_1 \texttt{ then } f_2 \texttt{ else } f_3 \texttt{ fi} \end{aligned}$$

A sequence of such expressions $\bar{f}$ can be substituted for a corresponding sequence of logical variables $\bar{z}$ of exactly the same type in the specification of the body S. Without these substitutions one cannot prove anything about the local state of the caller after the method invocation.

Next, we analyze reasoning about method invocations that are dynamically bound to an implementation like in the statement $S \equiv y = u.m(e_1, \ldots, e_n)$. For this purpose we first define some abbreviations.

Firstly, we formalize the set of classes that provide an implementation of a particular method. Assume that $\mathsf{methods}(C)$ denotes the set of method identifiers for which an implementation is given in class C. The function impl yields the class that provides the implementation of a method m for objects of a particular class. It is defined as follows.

$$\mathsf{impl}(C)(m) = \begin{cases} C & \text{if } m \in \mathsf{methods}(C) \\ \mathsf{impl}(F_{\lhd}(C))(m) & \text{otherwise} \end{cases}$$

We can generalize the above definition to get all implementations that are relevant to a particular domain. This results in the following definition.

$$\mathsf{impls}(C)(m) = \{C' \in \mathcal{C} | \mathsf{impl}(C'')(m) = C' \text{ for some class } C'' \text{ with } C'' \preceq C\} \ .$$

Thus the set $\mathsf{impls}([\![u]\!])(m)$ contains all classes that provide an implementation of method m that might be bound to the call $u.m(e_1, \ldots, e_n)$.

Another important issue when reasoning about methods calls is which classes inherit a particular implementation of a method. For that reason we consider the subclasses of a class C that override the implementation given in class C. We denote this set by $\mathsf{overrides}(C)(m)$. We have $C' \in \mathsf{overrides}(C)(m)$ if C' is a proper subclasses of C with $m \in \mathsf{methods}(C')$ and there does not exist another proper subclass C'' of C such that $C' \prec C''$ and $m \in \mathsf{methods}(C'')$. With this definition we can formulate the condition for an implementation of m in class C to be bound to a method call $u.m(e_1, \ldots, e_n)$. It is

$$u \texttt{ instanceof } C \wedge \neg u \in \mathsf{overrides}(C)(m) \ ,$$

where the latter clause abbreviates the conjunction

$$\bigwedge_{C' \in \mathsf{overrides}(C)(m)} \neg(u \texttt{ instanceof } C') \ .$$

We now have all building blocks for reasoning about a specification of the form $\{P\}\ y = u.m(e_1, \ldots, e_n)\ \{Q\}$. Assume that $\mathsf{impls}([\![u]\!])(m) = \{C_1, \ldots, C_k\}$. Let $\{\ S_i \texttt{ return } e_i\ \}$ be the body of the implementation of method m in class C_i, for $i = 1, \ldots, k$, and let $\bar{u_i}$ be its formal parameters. To derive a specification for $y = u.m(e_1, \ldots, e_n)$ we have to prove that for each implementation S_i a specification $B_i \equiv \{P_i \wedge I_i\} S_i \{Q_i[e_i/\texttt{return}_i]\}$ holds. Moreover, this specification should satisfy certain restrictions. First of all, the assertions P_i and Q_i must satisfy the same conditions as the assertions P and Q in the rule OMI. The assertion I_i is similar to the assertion I in that rule. Secondly, the preconditions of the implementations must be implied by the precondition of the call. That is, we must prove the following implications.

$$P \wedge u \texttt{ instanceof } C_i \wedge \neg u \in \mathsf{overrides}(C_i)(m) \\ \rightarrow P_i[(C_i)u, \bar{e}/\texttt{this}, \bar{u}_i][\bar{f}/\bar{z}] \qquad (\vec{P_i})$$

Similarly, we have to check wether the postconditions of the implementations imply the postcondition of the call. This requires proving the following formulas.

$$Q_i[(C_i)u/\texttt{this}][\bar{f}/\bar{z}] \rightarrow Q[\texttt{return}_i/y] \qquad (\vec{Q}_i)$$

The rule for dynamically-bound method invocations (DMI) then simply says that all given implications should hold and, moreover, we have to derive the specifications of the bodies.

$$\frac{\vec{P}_1, \ldots, \vec{P}_k \qquad B_1, \ldots, B_k \qquad \vec{Q}_1, \ldots, \vec{Q}_k}{\{P\}\ y = u.m(e_1, \ldots, e_n)\ \{Q\}} \qquad \text{(DMI)}$$

The generalization of the rule for non-recursive method invocations to one for recursive and even mutually recursive method invocations is a variant of the classical recursion rule. The idea behind the classical rule is to prove correctness of the specification of the body of the call on the assumption that the method call satisfies its specification. Our rule for mutually recursive method invocations (MRMI) allows both dynamically bound method invocations and calls to overridden methods in the recursion chain. To enable this it combines the rules (OMI) and (DMI). The outline of the rule is as follows.

$$\frac{F_1, \ldots, F_r \vdash \bar{B}_1, \ldots, \bar{B}_r \qquad \bar{P}_1, \ldots, \bar{P}_r \qquad \bar{Q}_1, \ldots, \bar{Q}_r}{F_1} \qquad \text{(MRMI)}$$

The formulas $F_1, \ldots, F_r$ are the specifications of the calls that occur in the recursion chain. That is, we require that each F_j is a correctness formula about a method invocation. As a naming convention, we assume that each F_j is of the form

$$\{P_j\}\ y_j = u_j.m_j(e^j_1, \ldots, e^j_{n_j})\ \{Q_j\} \text{ or } \{P_j\}\ y_j = \texttt{super}.m_j(e^j_1, \ldots, e^j_{n_j})\ \{Q_j\}\ .$$

The formulas $\bar{B}_j$, for $j = 1, \ldots, r$, denote sequences of correctness formulas about all possible implementations of the call in F_j. $\bar{B}_j$ contains only one element if F_j concerns a call to an overridden method. The sequences $\bar{P}_j$ and $\bar{Q}_j$ are the corresponding compatibility checks for the pre- and postconditions. Each element in $\bar{P}_j$ and $\bar{Q}_j$ corresponds to the element at the same position in F_j.

Let us first consider the case where F_j is of the form

$$\{P_j\}\ y_j = u_j.m_j(e^j_1, \ldots, e^j_{n_j})\ \{Q_j\}\ .$$

Assume that $\mathsf{impls}([\![u_j]\!])(m_j) = \{C_1, \ldots, C_{k_j}\}$. Let $\{\ S_i\ \texttt{return}\ e_i\ \}$ be the body of the implementation of method m_j in class C_i, for $i = 1, \ldots, k_j$, and let $\bar{u}_i$ be its formal parameters. Then $\bar{B}_j$ is the sequence containing, for $i = 1, \ldots, k_j$, the correctness formulas $\{P'_i \wedge I_i\}S_i\{Q'_i[e_i/\texttt{return}_i]\}$. The formula $\bar{P}_j$ is the sequence containing, for $i = 1, \ldots, k_j$, the implications

$$P_j \wedge u_j\ \texttt{instanceof}\ C_i \wedge \neg u_j \in \mathsf{overrides}(C_i)(m_j) \\ \rightarrow P'_i[(C_i)u_j, \bar{e}_j/\texttt{this}, \bar{u}_i][\bar{f}/\bar{z}_i]\ .$$

And finally, $\bar{Q}_j$ is the sequence containing, for $i = 1, \ldots, k_j$, the implications

$$Q'_i[(C_i)u_j/\mathtt{this}][\bar{f}/\bar{z}_i] \rightarrow Q_j[\mathtt{return}_i/y_j] \ .$$

On the other hand, if F_j is of the form $\{P_j\}\ y_j = \mathtt{super}.m_j(e^j_1, \ldots, e^j_{n_j})\ \{Q_j\}$ we can statically determine to which implementation this call is bound. Suppose that it is bound to the implementation $m_j(u_1, \ldots, u_n)\{\ S\ \mathtt{return}\ e\}$ in class C. Then the sequence $\bar{B}_j$ contains only the formula $\{P'_j \wedge I_j\}\ S\ \{Q'_j[e/\mathtt{return}_j]\}$. The compatibility check $\bar{P}_j$ is $P_j \rightarrow P'_j[(C)\mathtt{this}, \bar{e}_j/\mathtt{this}, \bar{u}_j][\bar{f}_j/\bar{z}_j]$ and $\bar{Q}_j$ is $Q'_j[(C)\mathtt{this}/\mathtt{this}][\bar{f}_j/\bar{z}_j] \rightarrow Q_j[\mathtt{return}_j/y_j]$. In the formula $\bar{P}_j$, $\bar{e}_j = e^j_1, \ldots, e^j_{n_j}$ and $u_i = u_1, \ldots, u_n$.

7 Conclusions

The main result of this paper is a syntax-directed Hoare logic for a language that has all standard object-oriented features. The logic extends our work as presented in [7, 8] by covering inheritance, subtyping and dynamic binding. We prove that the proposed Hoare logic is (relatively) complete in the full paper [3].

In recent years, several Hoare logics for (sequential) fragments of object-oriented languages, in particular Java, were proposed. However, the formal justification of existing Hoare logics for object-oriented languages is still under investigation. For example, completeness is still un open question for many Hoare logics in this area (see, e.g., [1, 9]).

However, in [10], a Hoare logic for a substantial sequential subset of Java is proved complete. This Hoare logic formalizes correctness proofs directly in terms of a semantics of the subset of Java in Isabelle/HOL. Higher order logic is used as specification language. As observed by the author this results in a serious discrepancy between the abstraction level of the Hoare logic and the programming language.

We are currently putting the finishing touch to a tool that enables the application of our logic to larger test-cases. It supports the annotation of programs, fully automatically computes the resulting verification conditions, and feeds them to a theorem prover. We aim to make this tool publicly available soon.

Checking the verification conditions is in general not decidable. However, we plan to investigate the isolation of a decidable subset of the present assertion language which would still allow, for example, aliasing analysis. Future work also includes the integration of related work on reasoning about abrupt termination [11] and concurrency in an object-oriented setting [12].

Finally, we would like to give a compositional formulation of the logic presented in this paper which will be based on invariants that specify the externally observable behavior of the objects in terms of the send and received messages. We envisage the use of temporal logics as described in [13] for the formulation of such invariants.

References

1. Poetzsch-Heffter, A., Müller, P.: A programming logic for sequential Java. In Swierstra, S.D., ed.: ESOP '99. Volume 1576 of LNCS. (1999) 162–176
2. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12** (1969) 576–580
3. Pierik, C., de Boer, F.S.: A syntax-directed Hoare logic for object-oriented programming concepts. Technical Report UU-CS-2003-010, Institute of Information and Computing Sciences, Utrecht University, The Netherlands (2003) Available at http://www.cs.uu.nl/research/techreps/UU-CS-2003-010.html.
4. Kowaltowski, T.: Axiomatic approach to side effects and general jumps. Acta Informatica **7** (1977) 357–360
5. Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison-Wesley (1996)
6. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Department of Computer Science, Iowa State University (2003)
7. de Boer, F.S.: A wp-calculus for OO. In Thomas, W., ed.: FoSSaCS '99. Volume 1578 of LNCS. (1999) 135–149
8. de Boer, F., Pierik, C.: Computer-aided specification and verification of annotated object-oriented programs. In Jacobs, B., Rensink, A., eds.: FMOODS V, Kluwer Academic Publishers (2002) 163–177
9. Reus, B., Wirsing, M., Hennicker, R.: A Hoare calculus for verifying Java realizations of OCL-constrained design models. In Hussmann, H., ed.: FASE 2001. Volume 2029 of LNCS. (2001) 300–317
10. von Oheimb, D.: Hoare logic for Java in Isabelle/HOL. Concurrency and Computation: Practice and Experience **13** (2001) 1173–1214
11. Huisman, M., Jacobs, B.: Java program verification via a Hoare logic with abrupt termination. In Maibaum, T., ed.: FASE 2000. Volume 1783 of LNCS. (2000) 284–303
12. Abraham-Mumm, E., de Boer, F., de Roever, W., Steffen, M.: Verification for Java's reentrant multithreading concept. In: FoSSaCS '02. Volume 2303 of LNCS. (2002) 5–20
13. Distefano, D., Katoen, J.P., Rensink, A.: On a temporal logic for object-based systems. In Smith, S.F., Talcott, C.L., eds.: FMOODS III, Kluwer Academic Publishers (2000) 305–326

Inheritance of Temporal Logic Properties

Heike Wehrheim

Universität Oldenburg, Fachbereich Informatik,
26111 Oldenburg, Germany
wehrheim@informatik.uni-oldenburg.de

Abstract. Inheritance is one of the key features for the success of object-oriented languages. Inheritance (or specialisation) supports incremental design and re-use of already written specifications or programs. In a formal approach to system design the interest does not only lie in re-use of class definitions but also in re-use of *correctness proofs*. If a provably correct class is specialised we like to know those correctness properties which are preserved in the subclass. This can avoid re-verification of already proven properties and may thus substantially reduce the verification effort.
In this paper we study the question of inheritance of correctness properties in the context of state-based formalisms, using a temporal logic (CTL) to formalise requirements on classes. Given a superclass and its specialised subclass we develop a technique for computing the set of formulas which are preserved in the subclass. For specialisation we allow addition of attributes, modification of existing as well as extension with new methods.

1 Introduction

Object-oriented languages nowadays play a major role in software development. This has even further increased with the advent of component-based programming. Object-orientation supports encapsulation (of data and operations within a class) and re-use of already written specifications or code. The latter aspect is (mainly) achieved by the concept of *inheritance* or *specialisation* allowing to derive new classes from existing ones.

In a formal approach to software development classes (or systems) are specified using a formal specification language. A formal specification having a precisely defined semantics opens the possibility of *proving correctness* of the design with respect to certain properties. Ideally, the design should be complemented by a number of formally stated requirements which the system is supposed to fulfill. In this context the concept of specialisation poses a new question: besides re-using specifications can we also re-use correctness proofs? A positive answer to this question would save us from a large verification effort: we would not have to redo all the correctness proofs. Within the area of program verification, especially of Java programs, this question has already been tackled by a number of researchers [9, 12, 8]. In these approaches correctness properties are mainly formulated in Hoare logic, and the aim is to find proof rules which help to deduce subclass properties from superclass properties. In order to get correctness of these rules it is required that the subclass is a *behavioural subtype* [10] of the superclass. This assumption is also the basis of [15] which studies preservation of properties in an event-based setting with correctness requirements formulated as CSP processes. In this paper we look at inheritance of properties to specialised classes form another point of view. Classes are not written in a programming language or as a CSP process but are defined in a general (state-based) mathematical framework. Correctness requirements on classes are formalised in a temporal logic (CTL). The major difference to previous work in this area is, however, that we do not restrict our considerations to specific subclasses (namely subtypes), but start with an arbitrary subclass and *compute* the set of properties which are preserved.

The class descriptions we employ describe *active* objects, providing as well as refusing services (method invocations) at certain points in time. To achieve this, every method is characterised by a *guard* defining its applicability and an *effect* defining the effect of its execution on attributes. A (passive) class which never refuses any call can also easily be described by setting all guards to *true*. For convenience of reading we will write class specifications in a formalism close to Object-Z [13], however, will work with a more general, formalism-independent definition of classes. There are two kinds of properties we are interested in (manifested in the choice of atomic propositions): first, properties over values of attributes, e.g. class invariants, and second, properties about the availability of services, e.g. that a certain method will always be executable[1].

Specialisation is not defined as an *operator* in a language but as a general relationship between class definitions[2]. Specialisation allows the addition of attributes, the modification of existing methods and the extension with new methods. It does not allow deletion of attributes or methods which would also be rather unusual for an inheritance operator.

The basic question we investigate in this paper can thus be formulated as follows: given two class definitions A and C, where C is a specialisation of A, which CTL-formulas holding for A hold for C as well? In general, a specialised class may inherit none of A's properties since it is possible to modify all methods and thus completely destroy every property of the superclass. Hence it is necessary to find out which modifications of methods (or extensions with new methods) influence the holding of CTL formulas. For this, we use a technique close to the *cone of influence reduction* technique used in hardware verification [2]. For every modified or new method we compute its *influence set*, i.e. the set of variables it directly (or indirectly) influences. Atomic propositions depending on variables in this influence set will thus potentially hold at different states in C than in A. Therefore formulas over atomic propositions of the influence set might not be inherited from A to C. All formulas independent of the in-

[1] Since methods are guarded, they can in general be refused in some states.

[2] Depending on the concrete formalism employed, inheritance might induce a specialisation relationship among classes in the sense used here.

fluence set are, however, preserved. This result is proven by showing that the two classes are *stuttering equivalent* with respect to the set of unchanged atomic propositions. Stuttering equivalence has been introduced by [1] for comparing Kripke structures according to the set of CTL (CTL*) formulas they fulfill. There are two different variants of stuttering equivalence (divergence-blind and divergence-sensitive) which correspond to two different interpretations of CTL [4]. We choose only one of them and will argue why this one is appropriate for our study of property inheritance and why the corresponding CTL interpretation might be more adequate for classes.

The paper is structured as follows. In the next section we discuss the basics necessary for formulating the result. We introduce the logic CTL, formalise class definitions and supply them with a semantics by assigning a Kripke structure to every class. We furthermore define divergence-blind stuttering equivalence and state the result that stuttering equivalent Kripke structures fulfill the same set of CTL-X (without Next) formulas. Section 3 starts with defining and explaining the influence set of methods. It furthermore states and proves the main result about inheritance of temporal logic properties over variables not in the influence set. Some examples further illustrate the result. Section 4 concludes.

2 Background

In this section we introduce the definitions necessary for our study of property inheritance. We describe class definitions and give an example of a class which we later use when looking at properties and specialisations. Furthermore we define the syntax and semantics of the logic CTL and state a result relating stuttering equivalence and CTL formulas.

2.1 Classes

Classes consist of attributes (or variables) and methods to operate on attributes. Methods may have input parameters and may return values, referred to as output parameters. We assume variables and input/output parameters to have values from a global set D. A *valuation* of a set of variables V is a mapping from V to D, we let $R_V = \{\rho : V \to D\}$ stand for the set of all valuations of V, the set of valuations of input parameters *Inp* and output parameters *Out* can be similarly defined.

A class is thus characterised by

- A set of *attributes* (or variables) V,
- an *initial valuation* of V to be used upon construction of objects: $I : V \to D$, and
- a set of methods (names) M with input and output parameters from a set of inputs *Inp* and a set of outputs *Out*. For simplicity we assume *Inp* and *Out* to be global. Each $m \in M$ has a guard $\mathit{guard}_m : R_V \times R_{\mathit{Inp}} \to \mathbb{B}$ ($\mathbb{B}$ are booleans) and an effect $\mathit{effect}_m : R_V \times R_{\mathit{Inp}} \to R_V \times R_{\mathit{Out}}$. The guard specifies the states in which the method is executable and the effect determines the outcome of the method execution.

The semantics of a class is defined in terms of *Kripke structures*, the standard model on which temporal logics are interpreted. A Kripke structure is similar to a transition system.

Definition 1. *Let AP be a nonempty set of atomic propositions. A* Kripke structure *$K = (S, S_0, \longrightarrow, L)$ over AP consists of a finite set of states S, a set of initial states $S_0 \subseteq S$, a transition relation $\longrightarrow \subseteq S \times S$ and a labeling function $L : S \rightarrow 2^{AP}$.*

Note that we do not require totality of the transition relation, i.e. there may be states $s \in S$ such that there is no s' with $s \longrightarrow s'$.

The basic properties we like to observe about a class (or an object) are its state and the availability of methods. Thus the atomic propositions AP_A that we consider for a class $A = (V, I, M, (guard_m)_{m \in M}, (effect_m)_{m \in M})$ are

- $v = d$, $v \in V, d \in D$ and
- $enabled(m)$, $m \in M$.

We assume method execution to be atomic, i.e. we do not distinguish beginnings and ends of methods and do not allow concurrent access to an object.

Definition 2.
The semantics of (an object of) a class $A = (V, I, M, (guard_m)_{m \in M}, (effect_m)_{m \in M})$ is the Kripke structure $K = (S, S_0, \longrightarrow, L)$ over AP_A with

- $S = R_V$,
- $S_0 = I$,
- $\longrightarrow = \{(s, s') \mid \exists\, m \in M, \exists\, \rho_{in} \in R_{Inp}, \rho_{out} \in R_{Out} : guard_m(s, \rho_{in})$
  $\wedge\; effect_m(s, \rho_{in}) = (s', \rho_{out})\}$,
- $L(s) = \{v = d \mid s(v) = d\} \cup \{enabled(m) \mid \exists\, \rho_{in} \in R_{Inp} : guard_m(s, \rho_{in})\}$.

Since the atomic propositions do not refer to inputs and outputs of methods, they are not reflected in the semantics. The following example gives a class definition of a simple bank account with methods for disposal and withdrawal. To enhance readability we have chosen a presentation which is close to Object-Z. The first box (schema) describes the attributes (with their domains, N being a constant), the *Init* schema defines the initial valuation and schemas labeled *guard* and *effect* define guards and effects of the methods. Primed variables refer to attribute values in the next state, the notation *var*? describes an input parameter and $\Delta(\ldots)$ denotes the set of variables which are changed upon execution of particular methods.

BankAccount

$money : [-N..N]$

INIT

$money = 0$

guard_Deposit

$amount? : [1..N]$

$true$

effect_Deposit

$\Delta(money)$

$amount? : [1..N]$

$money' = money + amount?$

guard_Withdraw

$amount? : [1..N]$

$money \geq amount?$

effect_Withdraw

$\Delta(money)$

$amount? : [1..N]$

$money' = money - amount?$

Figure 1 shows the Kripke structure (without L) of class *BankAccount*. The numbers indicate the values of attribute *money*. All states with $money < 0$ are unreachable. The upper arrows correspond to executions of *Deposit*, the lower to those of *Withdraw*.

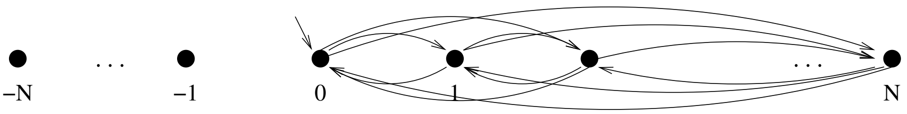


Fig. 1. Kripke structure of class *BankAccount*.

2.2 CTL

The temporal logic that we use for specifying properties of classes is CTL [5]. The temporal operators of CTL consist of a path quantifier (E for existential and A for universal quantification) plus operators for expressing eventuality (F), globally (G), next state (X), and until (U).

Definition 3. *The set of CTL formulas over AP is defined as the smallest set of formulas satisfying the following conditions:*

- $p \in AP$ *is a formula,*
- *if* φ_1, φ_2 *are formulas, so are* $\neg\varphi_1$ *and* $\varphi_1 \vee \varphi_2$*,*
- *if* φ *is a formula, so are* $EX\varphi$*,* $EF\varphi, EG\varphi$*,*
- *if* φ_1, φ_2 *are formulas, so is* $E[\varphi_1\ U\ \varphi_2]$*.*

As usual, other operators can be defined as abbreviations, e.g. $true = (p \vee \neg p)$ for some arbitrary proposition p, $false = \neg true$, $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $AG\ \varphi = \neg EF\ \neg\varphi$ and $AX\ \varphi = \neg EX\ \neg\varphi$[3]. CTL-X is the set of CTL formulas without the next-operators EX and AX.

For the interpretation of CTL formulas we look at *paths* of the Kripke structure.

[3] The formulae $AF\varphi_2$ and $A[\varphi_1\ U\ \varphi_2]$ have been omitted here since, under the interpretation given in Definition 5, they both coincide with φ_2.

Definition 4. *Let $K = (S, S_0, \rightarrow, L)$ be a Kripke structure over AP.*

- *A* path *is a nonempty finite or infinite sequence of states $\pi = s_0 s_1 s_2 \dots$ such that $s_i \rightarrow s_{i+1}$ holds for all $i > 0$ (in case of finite paths up to some n). For a path $\pi = s_0 s_1 s_2 \dots$ we write $\pi[i]$ to stand for s_i. We define the* length *of a finite path $\pi = s_0 \dots s_{n-1}$, $|\pi|$, to be n and set the length of infinite paths to ∞.*
- *The set of paths starting in $s \in S$ is*

$$paths(s) = \{s_0 s_1 s_2 \dots \mid s_0 s_1 s_2 \dots \text{ is a path and } s_0 = s\}$$

Note that paths need not be maximal or infinite. In this, we deviate from the standard interpretation of CTL. Usually, only infinite paths are considered, whereas we also look at prefixes of such infinite paths. The interpretation we give below has, however, also been considered by [6] and [4].

Definition 5. *Let $K = (S, S_0, \rightarrow, L)$ be a Kripke structure and φ a CTL formula, both over AP. K satisfies φ ($K \models \varphi$) iff $K, s_0 \models \varphi$ holds for all $s_0 \in S_0$, where $K, s \models \varphi$ is defined as follows:*

- $K, s \models p$ *iff* $p \in L(s)$,
- $K, s \models \neg\varphi$ *iff not* $K, s \models \varphi$,
- $K, s \models \varphi_1 \vee \varphi_2$ *iff* $K, s \models \varphi_1$ *or* $K, s \models \varphi_2$,
- $K, s \models EX\ \varphi$ *iff* $\exists\, \pi \in path(s), |\pi| > 1 \wedge K, \pi[1] \models \varphi$,
- $K, s \models EF\ \varphi$ *iff* $\exists\, \pi \in path(s), \exists\, k \geq 0 : K, \pi[k] \models \varphi$,
- $K, s \models EG\ \varphi$ *iff* $\exists\, \pi \in path(s), \forall\, k \geq 0, k < |\pi| : K, \pi[k] \models \varphi$,
- $K, s \models E[\varphi_1\ U\ \varphi_2]$ *iff* $\exists\, \pi \in path(s), \exists\, k \geq 0 : K, \pi[k] \models \varphi_2$ *and* $\forall\, j, 0 \leq j < k : K, \pi[j] \models \varphi_1$.

This interpretation is more convenient for our study in two respects. One reason will be discussed later when we look at preserved properties: Under the standard interpretation fewer properties are inherited to specialised classes. For the second point consider again the class specification *BankAccount*. Its Kripke structure K satisfies the following properties:

S1 $AG\ (money \geq 0)$,
S2 $AG\ (enabled(Deposit))$,
L $AG\ EF\ (enabled(Withdraw))$.

The formula $AF\ (money > 0)$ does, however, not hold for the Kripke structure. This is due to the non-standard semantics of CTL given here since it does also consider non-maximal paths. The path consisting of just the initial state does not satisfy the formula; the formula would hold if we restrict paths to maximal (infinite) ones. Since the execution of methods, however, depends on calls of methods from the outside and is not under control of the class, we actually can never be sure that steps to next states are taken. Thus, an object of class *BankAccount* might actually never be used and thus we might never reach a state with $money > 0$.

2.3 Stuttering Equivalence

The last part of the background section concerns the result relating stuttering equivalence with CTL formulas. Stuttering equivalence will be the relation used to relate specialised classes with superclasses. Stuttering equivalence is the state-based analog of branching bisimulation [14].

Definition 6. *Let* $K_i = (S_i, S_{0,i}, \rightarrow_i, L_i)$*,* $i = 1, 2$*, be Kripke structures over* AP_1, AP_2*, respectively.* K_1 *and* K_2 *are* divergence blind stuttering equivalent *with respect to a set of atomic propositions* $AP \subseteq AP_1 \cap AP_2$ *(*$K_1 \approx_{AP} K_2$*) iff there is a relation* $B \subseteq S_1 \times S_2$ *which satisfies the following conditions:*

1. $\forall\, s_1 \in S_{0,1}\, \exists\, s_2 \in S_{0,2} : (s_1, s_2) \in B$, *and vice versa,*
   $\forall\, s_2 \in S_{0,2}\, \exists\, s_1 \in S_{0,1} : (s_1, s_2) \in B$,

and for all $(r, s) \in B$

2. $L_1(r) \cap AP = L_2(s) \cap AP$,
3. $r \rightarrow r' \Rightarrow$
   $\exists\, s_0, s_1, \ldots, s_n, n \geq 0$ *such that* $s_0 = s, \forall\, 0 \leq i < n : s_i \rightarrow s_{i+1}$
   $\wedge\, (r, s_i) \in B$, *and* $(r', s_n) \in B$, *and vice versa,*
   $s \rightarrow s' \Rightarrow$
   $\exists\, r_0, r_1, \ldots, r_n, n \geq 0$ *such that* $r_0 = r, \forall\, 0 \leq i < n : r_i \rightarrow r_{i+1}$
   $\wedge\, (r_i, s) \in B$, *and* $(r_n, s') \in B$.

Stuttering equivalence allows stuttering steps during the simulation of one step of K_1 by K_2 and vice versa. All the intermediate states in the stuttering step (e.g. $s_0, \ldots s_{n-1}$) still have to be related to the starting state of the other structure, i.e. the atomic propositions from AP may not change during stuttering. Since we allow stuttering a property holding in the *next* state of K_1 possibly does not hold in the next but only in later states of K_2. Thus concerning CTL formulas, stuttering equivalent Kripke structures only satisfy the same set of CTL-X formulas.

Theorem 1. *Let* K_1, K_2 *be two Kripke structures such that* $K_1 \approx_{AP} K_2$*, and let* φ *be a CTL-X formula over* AP*. Then the following holds:*

$$K_1 \models \varphi \quad \textit{iff} \quad K_2 \models \varphi$$

Proof. See [1, 4].

3 Inheriting Properties

Usually, inheritance of properties is studied for subclasses which are behavioural subtypes of their superclasses. Here we pursue a different line of thought. We allow to arbitrarily add new methods as well as change old methods and impose no a priori restrictions on these extensions and modifications. Since this may in general lead to a specialised class in which lots of properties of the superclass do not hold anymore, we have to compute the set of formulas which are preserved. Instead of restricting the subclasses considered, as is usually done, we thus restrict the set of formulas.

First, we have to define what it means for a class to be a *specialisation* of another class. Instead of defining a particular operator we simply define a relation. In the following we always assume that the classes A and C have components $(V_A, I_A, M_A, (guard_m^A)_{m \in M_A}, (effect_m^A)_{m \in M_A})$ and $(V_C, I_C, M_C, (guard_m^C)_{m \in M_C}, (effect_m)_{m \in M_C}^C)$, respectively.

Definition 7. *Let A and C be classes. C is a* specialisation *of A if $V_A \subseteq V_C$, $M_A \subseteq M_C$ and $I_C \mid_{V_A} = I_A$.*

In general, a subclass derived from a superclass by inheritance will be a specialisation of the superclass. C may change method definitions as well as introduce new methods: the changed methods are those methods m in M_A for which either $guard_m^A \neq guard_m^C$ or $effect_m^A \neq effect_m^C$, the new methods are those in $M_C \setminus M_A$. To keep matters simple we assume that the subclass does not change initial values of old attributes.

For computing the set of preserved formulas we have to find out which atomic propositions are affected by the modifications or extensions made in the specialised class. For this we need to compute the *influence set* of methods, i.e. the set of variables that may directly or indirectly be affected by method execution. First, we define the set of variables that a method modifies, uses and depends on. Intuitively, a method depends on a variable if the evaluation of its guard is affected by the value of the variable; it uses a variable if the value of the variable is used to determine output or changes to the attributes; and it modifies a variable if the value of the variable is changed upon execution of the method.

Note that in the definition below, $first(effect_m(..., ...))$ refers to the new state of attributes after execution of a method m (the first component of a pair from $R_V \times R_{Out}$).

Definition 8. *Let $A = (V, I, M, (guard_m)_{m \in M}, (effect_m)_{m \in M})$ be a class.*

A method $m \in M$ modifies *a variable $v \in V$ if there are $\rho \in R_V, \rho_{in} \in R_{Inp}$ such that $first(effect_m(\rho, \rho_{in}))(v) \neq \rho(v)$.*

Execution of $m \in M$ depends on *$v \in V$ if there are $\rho_1, \rho_2 \in R_V, \rho_{in} \in R_{Inp}$ such that $\rho_1 \mid_{V \setminus \{v\}} = \rho_2 \mid_{V \setminus \{v\}}$ and $guard_m(\rho_1, \rho_{in}) \neq guard_m(\rho_2, \rho_{in})$.*

A method $m \in M$ uses *$v \in V$ if there are $\rho_1, \rho_2 \in R_V, \rho_{in} \in R_{Inp}$ such that $\rho_1 \mid_{V \setminus \{v\}} = \rho_2 \mid_{V \setminus \{v\}}$ and $effect_m(\rho_1, \rho_{in}) \neq effect_m(\rho_2, \rho_{in})$.*

We let $mod^A(m)$ denote the set of variables m modifies in A, $depends^A(m)$ the set it depends on, and $uses^A(m)$ the set of variables it uses. When clear from the context we omit the index for the class. For the formalism used in the *BankAccount* example these sets can be (over-)approximated by using the following rules: $mod(m)$ is the set of variables appearing in the Δ-list of $effect_m$, $uses(m)$ are all variables appearing in $effect_m$, and $depends(m)$ are those appearing in $guard_m$. Thus we for instance have $mod(Withdraw) = \{money\}$ and $depends(Deposit) = \varnothing$. Over-approximation does not invalidate the result about preservation of properties proven below.

There is a tight connection between the variables a method uses or depends on and its guard and effect, as expressed by the next proposition.

Proposition 1. *Let $V' \subseteq V$ be a set of variables, $s, s' \in R_V$ two states such that $s \mid_{V'} = s' \mid_{V'}$ and let $m \in M$ be a method.*

If $depends(m) \subseteq V'$ then $\forall \rho_{in} \in R_{Inp} : guard_m(s, \rho_{in}) \Leftrightarrow guard_m(s', \rho_{in})$.

If $uses(m) \subseteq V'$ then $\forall \rho_{in}, \rho_{out}, \rho'_{out}$ such that $effect_m(s, \rho_{in}) = (t, \rho_{out})$, $effect_m(s', \rho_{in}) = (t', \rho'_{out})$, we get $t \mid_{V'} = t' \mid_{V'}$ and $\rho_{out} = \rho'_{out}$.

The set of methods which potentially influence the holding of formulas are those changed in the specialised class C (compared to the superclass A) or new in C. In the following we denote this set of methods by N. We are interested in the effect these methods in N may have wrt. the atomic propositions holding in (the Kripke structure of) A. If a method in N modifies a variable which is used in a guard of another method in A, then in C this method might be enabled at different states and its execution might produce different results. This leads to the following definition of *influence set.*

Definition 9. *Let A and C be classes such that C is a specialisation of A. The* influence set *of a set of methods $N \subseteq M_C$, $Infl^{A,C}(N)$, is the smallest set of variables satisfying the following two conditions:*

1. *$\forall m \in N : mod^A(m) \cup mod^C(m) \subseteq Infl^{A,C}(N)$,*
2. *if $v \in Infl^{A,C}(N)$ and there is some $m' \in M_A$ such that $v \in depends^A(m')$ or $v \in uses^A(m')$, then $mod^A(m') \cup mod^C(m') \subseteq Infl^{A,C}(N)$.*

Note that in 2. we refer to class A only since A and C agree on methods outside of N anyway. In the sequel we always write $Infl(N)$ instead of $Infl^{A,C}(N)$. The influence set is recursively defined since modifications made to a variable may propagate to other variables. At the end of this section an example illustrating this propagation of changes can be found.

The influence set computation is similar to the cone of influence computation used as an abstraction technique in hardware verification [2]. However, while the cone of influence is computed for the set of variables under interest (and the remaining variables can then be abstracted away), we take the opposite view here: the influence set tells us which atomic propositions might *not* hold anymore, and the set of preserved formulas has to be restricted to the remaining atomic propositions.

Given a set of non-modified variables V' and non-modified methods M' we define the atomic propositions over V' and M' to be

$$\begin{aligned} AP_{V',M'} = &\{v' = d \in AP \mid v' \in V'\} \\ &\cup \{enabled(m') \in AP \mid m' \in M' \wedge depends(m') \subseteq V'\} \end{aligned}$$

Starting from a temporal logic formula over atomic propositions in A and having given a specialised class C, the set of non-modified variables and methods is thus $V' = V_A \setminus (N)$ and $M' = M_A \setminus N$ (where N is the set of methods modified or added in C).

The following theorem now states the main result of this paper. Concerning the atomic propositions over non-modified variables and methods, a class A and its specialisation C are stuttering equivalent.

Theorem 2. *Let A and C be classes such that C is a specialisation of A. Let N denote the set of methods changed in C or new in C, and set V' to $V_A \setminus \mathit{Infl}(N)$, M' to $M_A \setminus N$. Then the following holds:*

$$K_C \approx_{AP_{V',M'}} K_A$$

Intuitively, all steps corresponding to the execution of new or changed methods are now stuttering steps under the restricted set of atomic propositions.

Together with Theorem 1 this gives us the following corollary:

Corollary 1. *Let A and C be classes such that C is a specialisation of A. Let N denote the set of methods changed in C or new in C, and set V' to $V_A \setminus \mathit{Infl}(N)$, M' to $M_A \setminus N$.*
Let φ be a CTL-X formula over $AP_{V',M'}$. Then the following holds:

$$K_A \models \varphi \;\Leftrightarrow\; K_C \models \varphi$$

In particular, class C preserves *all* of A's properties if it does not change existing methods and new methods only modify new attributes, as expressed by the following corollary.

Corollary 2. *Let A and C be classes such that C is a specialisation of A. Let N denote the set of methods changed in C or new in C.*

If $N \subseteq M_C \setminus M_A$ and $\mathit{mod}^A(N) \cup \mathit{mod}^C(N) \subseteq V_C \setminus V_A$ then the following holds for all formulas $\varphi \in$ CTL-X over AP_{V_A,M_A}:

$$K_A \models \varphi \Leftrightarrow K_C \models \varphi$$

In this case, class C can also be seen as a behavioural subtype of A in the sense of [10] (extension rule).

Proof of Theorem 2. Let $K_A = (S_A, S_{0,A}, \rightarrow_A, L_A)$ and $K_C = (S_c, S_{0,C}, \rightarrow_C, L_C)$ be the Kripke structures of A and C. The relation showing divergence blind stuttering equivalence between K_A and K_C is $B = \{(s_A, s_C) \mid s_A\mid_{V'} = s_C\mid_{V'}\}$, where $V' = V_A \setminus \mathit{Infl}(N)$. We have to check conditions 1 to 3.

1. Holds since C is a specialisation of A and hence $I_C\mid_{V_A} = I_A$ which implies $I_C\mid_{V'} = I_A\mid_{V'}$.

Let $(s_A, s_C) \in B$.

2. Concerning $AP_{V',M'}$ the same set of atomic propositions hold in related states:

$$\begin{aligned}
L_A(s_A) \cap AP_{V',M'} &= \{v' = d \mid v' \in V', d \in D, s_A(v') = d\} \\
&\quad \cup \{\mathit{enabled}(m') \mid m' \in M', \mathit{depends}^A(m') \subseteq V' \\
&\qquad \wedge \exists\, \rho_{in} \in R_{\mathit{Inp}} : \mathit{guard}^A_{m'}(s_A, \rho_{in})\} \\
&= \{v' = d \mid v' \in V', d \in D, s_C(v') = d\} && (1) \\
&\quad \cup \{\mathit{enabled}(m') \mid m' \in M', \mathit{depends}^C(m') \subseteq V' \\
&\qquad \wedge \exists\, \rho_{in} \in R_{\mathit{Inp}} : \mathit{guard}^C_{m'}(s_C, \rho_{in})\} && (2) \\
&= L_C(s_C) \cap AP_{V',M'}
\end{aligned}$$

Line 1 holds since $s_A \mid_{V'} = s_C \mid_{V'}$ and line 2 holds since $\mathit{guard}^A_{m'} = \mathit{guard}^C_{m'}$ and thus also $\mathit{depends}^A(m') = \mathit{depends}^C(m')$.

3. Assume $s_A \rightarrow s'_A$. Then there is some $m \in M_A$, some $\rho_{in} \in R_{\mathit{Inp}}, \rho_{out} \in R_{\mathit{Out}}$ such that $\mathit{guard}^A_m(s_A, \rho_{in})$ and $\mathit{effect}^A_m(s_A, \rho_{in}) = (s'_A, \rho_{out})$. There are now three cases to consider: m is a method which is unchanged in C and a) m does only depend on and does only use variables in V', or b) m uses or depends on variables in $\mathit{Infl}(N)$, and third case $m \in N$. In the first case the transition in A is matched by a corresponding transition in C (the same method is executed), in the other two cases the A-step is a stuttering step; it essentially changes atomic propositions not in $AP_{V',M'}$.
   (a) $m \in M_A \setminus N$ and $\mathit{depends}^A(m) \cup \mathit{uses}^A(m) \subseteq V'$ (and thus $\mathit{depends}^C(m) \cup \mathit{uses}^C(m) \subseteq V'$): By Proposition 1 $\mathit{guard}^C_m(s_C, \rho_{in})$ and $\exists\, s'_C : s'_C \mid_{V'} = s'_A \mid_{V'}$ and $\mathit{effect}^C_m(s_c, \rho_{in}) = (s'_C, \rho_{out})$, i.e. $s_C \rightarrow s'_C$, and by definition of B, $(s'_A, s'_C) \in B$.
   (b) $m \in M_A \setminus N$ and $\mathit{depends}^A(m) \not\subseteq V'$ or $\mathit{uses}^A(m) \not\subseteq V'$:
   Then there is some $v \in \mathit{depends}^A(m)$ or $v \in \mathit{uses}^A(m)$ such that $v \in \mathit{Infl}(N)$. Since m is unchanged, we get by definition of the influence set $\mathit{mod}^A(m) \subseteq \mathit{Infl}(N)$. Hence $s_A \mid_{V'} = s'_A \mid_{V'}$ and therefore $(s'_A, s_C) \in B$ (the A-step is matched by 0 C-steps).
   (c) $m \in N$ (a changed method):
   Then $\mathit{mod}^A(m) \subseteq \mathit{Infl}(N)$ and we can apply the same reasoning as in the last case.
4. Reverse case: $s_C \rightarrow s'_C$. Then there is some $m \in M_C$, some $\rho_{in} \in R_{\mathit{Inp}}, \rho_{out} \in R_{\mathit{Out}}$ such that $\mathit{guard}^C_m(s_C, \rho_{in})$ and $\mathit{effect}^A_m(s_C, \rho_{in}) = (s'_C, \rho_{out})$. We essentially get the same three cases as before (except for the fact that in the third case we can now have a new as well as a changed method) and can therefore reason analogously. □

We illustrate the result by our bank account example. Below two specialisations of class *BankAccount* are given. Both are defined using inheritance which here simply means that all definitions of *BankAccount* (i.e. attributes and methods) also apply for the specialised classes. The first class adds two new attributes and one method to *BankAccount*.

The new method *AssignId* assigns an identification number to a bank account. Once this number has been assigned method *AssignId* is disabled. The new method does not change the old variable:

$$\mathit{Infl}^{BankAccount,BA1}(AssignId) = \{idAssigned, id\}$$

All of our properties S1, S2 and L are thus preserved.

*BA*1

inherits BankAccount

$id : ID$
$idAssigned : \mathbb{B}$

INIT
$\neg idAssigned$

guard_AssignId
$id? : ID$

$\neg idAssigned$

effect_AssignId
$\Delta(idAssigned)$
$id? : ID$

$idAssigned'$
$id' = id?$

Consider on the other hand the second specialised class *BA*2: It modifies method *Withdraw* which means that the definition of *Withdraw* in *BankAccount* is replaced by the new definition. In this bank account the user is allowed to overdraw his account (up to $-N$).

*BA*2

inherits BankAccount

modifies Withdraw

guard_Withdraw
$amount? : [1..N]$

$money - amount? \geq -N$

effect_Withdraw
$\Delta(money)$
$amount? : [1..N]$

$money' = money - amount?$

We have $\mathit{Infl}^{BankAccount,BA2}(Withdraw) = \{money\}$. Thus our theorem only tells us that S2 is preserved but it tells us nothing about S1 and L. While the liveness property L does still hold, S1 is invalidated in *BA*2.

A

$x, y, z : \mathbb{N}$

INIT
$x = y = z = 0$

*effect_op*1
$\Delta(x)$

$x' = x + z$

*effect_op*2
$\Delta(y)$

$y' = y + x$

The last (rather artifical) example shows why we need the inclusion of variables into the influence set which are only indirectly affected by a new or changed method. Consider the following class A with three attributes x, y and z, where x and y are modified by operations $op1$ and $op2$, respectively.
The property $AG\ (y = 0)$ holds for this class. Next we make a specialisation of the class which adds one new method modifying z.

C

inherits A

*effect_op*3

$\Delta(z)$

$z' = z + 1$

This method indirectly influences variable y: the value of y depends on x and x depends on z, $\mathit{mod}^C(op3) = \{z\}$, and $\mathit{Infl}^C(op3) = \{x, y, z\}$. Hence no formulas with atomic propositions talking about x, y or z are preserved, and it can in fact be seen that property $AG\ (y = 0)$ does not hold for class C anymore.

Finally, there is one issue which remains to be discussed: the choice for the specific non-standard interpretation of CTL. Usually, CTL is interpreted on maximal paths. Paths are always infinite which is achieved by requiring the transition relation $\longrightarrow$ to be total. In order to get a result similar to Theorem 1 in this case, an additional condition is needed in the definition of stuttering equivalence (see [3]): whenever two states r and s are related via B, r has *infinite stuttering* if and only if s has. Infinite stuttering means that there is an infinite path $rr_1r_2r_3\ldots$ from r such that all r_i are related with s. Intuitively this corresponds to an infinite sequence of method applications which do not change the atomic propositions under consideration. This extended definition is referred to as *divergence sensitive stuttering equivalence.* It is needed for preservation of CTL-X properties under the standard interpretation since for every path in r we need a corresponding path in s. If we only consider maximal paths we may lack a corresponding path when r has infinite stuttering but s does not. When using the non-standard interpretation a path may also consist of a single state.

In principle, we could have used this extended definition of stuttering equivalence together with the standard CTL semantics. This has, however, one significant drawback: the specialised class C then has to be divergence sensitive stuttering equivalent to the superclass A in order to inherit properties from A. This is in general not the case: for instance, a new method in C (corresponding to the stuttering steps) that may be executed infinitely often (e.g. when its guard is true) leads to infinite stuttering. Since this method is not present in A there may be no corresponding infinite stuttering in A. Thus, divergence-blind stuttering equivalence is more adequate for comparing specialised class with superclass. If nevertheless the standard interpretation should be used then one possibility is to add one extra method to every class. This method should always be enabled (guard = true) and should not change any variable. Thus every state has infinite stuttering and the necessary conditions are trivially fulfilled.

4 Conclusion

In this paper, we investigated the question of property inheritance from superclasses to specialised classes. Unlike other approaches we did not assume any specific relationship as for instance subtyping between superclass and subclass. Instead, we showed how to *compute* the set of preserved properties when given a class and its specialisation.

Related Work. The work most closest to ours, both from a technical point of view and from its overall aim, is that of cone-of-influence reduction in hardware [2] and program slicing [7, 11] in software verification or specification analysis. Both techniques start from a large system (circuit, program or specification) and apply the reduction technique to obtain a smaller one on which verification/analysis takes place. We start with the small system (class) and compute the influence set to find out which properties are preserved in the larger system.

The techniques used to prove the correctness of the main result are nevertheless quite similar: the cone-of-influence reduction technique shows bisimilarity of large and reduced system and thus preserves all CTL formulas; program slicing, as employed in [7], achieves a trace-based variant of stuttering equivalence between large and reduced system and thus preserves all LTL-X formulas.

Future Work. This work is only a first step towards a practical use of the computation of preserved properties. The most important extension concerns the consideration of more than one class. This can be done by combining the technique presented here with *compositional* verification techniques. In the setting of Object-Z this could for instance be the work of [16]. Another important issue is the actual computation of the influence set: in order to be able to apply our technique the influence set should be as small as possible and thus purely syntax-oriented computations might not be practical.

Acknowledgement

Thanks to one of the anonymous referees for its detailed comments which helped to improve the paper.

References

1. M.C. Browne, E.M. Clarke, and O. Grumberg. Characterising Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science*, 59:115–131, 1988.
2. E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.
3. D. Dams. Flat Fragments of CTL and CTL*: Separating the Expressive and Distinguishing Powers. *Logic Journal of IGPL*, 7(1):55–78, 1999.
4. Rocco De Nicola and Frits Vaandrager. Three logics for branching bisimulation. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 118–129. IEEE, Computer Society Press, 1990.

5. E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronisation skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.
6. E.A. Emerson and J. Srinivasan. Branching time temporal logic. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 123–171. Springer, 1988.
7. J. Hatcliff, M. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-order and Symbolic Computation.* To appear.
8. K. Huizing and R. Kuiper. Reinforcing fragile base classes. In A. Poetzsch-Heffter, editor, *Workshop on Formal Techniques for Java Programs, ECOOP 2001*, 2001.
9. G.T. Leavens and W.E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32:705–778, 1995.
10. B. Liskov and J. Wing. A behavioural notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811 – 1841, 1994.
11. T. Oda and K. Araki. Specification slicing in formal methods of software development. In *Proceedings of the Seventeenth Annual International Computer Software & Applications Conference*, pages 313–319. IEEE Computer Society Press, 1993.
12. A. Poetzsch-Heffter and J. Meyer. Interactive verification environments for object-oriented languages. *Journal of Universal Computer Science*, 5(3):208–225, 1999.
13. G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publisher, 2000.
14. Rob van Glabbeek and W. P. Weijland. Refinement in branching time semantics. In *Proc. IFIP Conference*, pages 613–618. North-Holland Publishing Company, 1989.
15. H. Wehrheim. Behavioural subtyping and property preservation. In S. Smith and C. Talcott, editors, *FMOODS'00: Formal Methods for Open Object-Based Distributed Systems.* Kluwer, 2000.
16. K. Winter and G. Smith. Compositional Verification for Object-Z. In D. Bert, J.P. Bowen, S. King, and M. Walden, editors, *ZB 2003: Formal Specification and Development in Z and B*, number 2651 in LNCS, pages 280–299. Springer, 2003.

Temporal Logic Based Static Analysis for Non-uniform Behaviours

Matthias Colin, Xavier Thirioux, and Marc Pantel

ENSEEIHT-IRIT
2, rue Camichel
31071 Toulouse Cedex
France
{colin,thirioux,pantel}@enseeiht.fr

Abstract. The main purpose of our work is the typing of concurrent, distributed and mobile programs based on the actor programming model, that is non-uniform behaviour concurrent objects communicating by asynchronous message passing. One of the key difficulties is to give a precise definition of "message not understood" errors in this context. In this paper, we investigate temporal logic and model-checking based technologies for an asynchronous message passing process calculi. We focus on non uniform input interfaces for processes, and then define a temporal logic tailored to their description and analyses. This logic deals with infinite-state systems, as mailboxes of actors are unbounded multisets of messages, but yet happens to be decidable. We use our logic to specify possible communication errors in actor-based programs in order to give precise and sound definition of type disciplines.

Introduction

The development of the telecommunication industry and the generalization of network use bring concurrent and distributed programming in the limelight. In that context, programming is a hard task and, generally, the resulting applications contain much more *bugs* than usual centralized software. As sequential object oriented programming is commonly accepted as a *good* way to build software, concurrent object oriented programming seems to be well-suited for programming distributed systems. Since non-determinism resulting from the unreliability of networks makes it difficult to validate any distributed functionality using informal approaches, our work is focused on applying formal type systems to improve concurrent object oriented programming.

To obtain widely usable tools, we have chosen to use the actor model proposed by Hewitt in [HBS73] and developed by Agha in [Agh86]. This model is based on a network of autonomous and cooperative agents (called actors), which encapsulate data and programs, communicating using an asynchronous point to point protocol. An actor stores each received message in a queue and when idle, processes the first message it can handle in this queue. Besides those conventions (which are also true for concurrent objects), an actor can dynamically change its interface. This property allows to increase or decrease the set of messages an actor may handle, yielding a more accurate programming model. This model, also known as concurrent objects with non uniform behaviour (or interface), has been adopted by the telecommunication industry for the development of distributed and concurrent applications for the Open Distributed Computing framework (ITU X901-X904) and the Object Description Language (TINA-C extension of OMG IDL with multiple interfaces). Due to behaviour change, it may happen that a message cannot be handled by its target in some execution path (a sequence of behaviours the actor can assume) and could be handled in some other path. Such messages are called orphan messages.

Type systems for concurrent objects and actors, with uniform or non-uniform behaviours, have been the subject of active research in the last years ([RV00], [Nie95], [Kob97], [Pun96], [NNS99], [FLMR97], [Bou97] and their more recent works). Two opposite approaches have been followed: explicit and implicit typing. Explicit types (see [RR01,CRR02]) may provide more precise information but are sometimes very hard to write for the programmer (they might be much more complex than the program itself). We advocate the use of implicit typing, i.e. type inference, as it is simpler to use.

In a first approach, our type systems were defined using Cap, an actor calculus derived from asynchronous π-calculus and Cardelli's Calculus of Primitive Objects (see [CPS96]). Two type systems were developed: the former (see [CPS97]) is based on usual object type abstractions and catches all functional and communication errors but only trivial orphans, the latter detects a large set of orphans but requires a much more complex type abstraction (see [CPDS99]).

Despite several unsuccessful attempts to provide formal definitions for the various kinds of orphan messages we are interested in ([CPDS99] and [DPCS00]), most of the time, the best definition we could give was: *orphan messages are the messages detected by our analyses* which was neither satisfying nor useful. The purpose of this paper is to present a decidable temporal logic dedicated to the specification of orphan messages which we use to compute various notions of input capacities for actors.

We initially give a short description of Cap, a primitive actor calculus used to define our static analyses.

1 Cap: A Primitive Actor Calculus

Various encodings of concurrent objects in the π-calculus or similar formalisms have been proposed [Wal95,PT97]. Message labels and actors mail addresses are usually both expressed using names. Therefore, the typing of encoded programs generally leads to type information which does not reflect the structure of the original program. The authors defined in [CPS96] the Cap calculus in order to overcome these constraints.

As in the π-calculus [Mil91] and in the ν-calculus [HT91] the basis of the calculus is the *name* representing the actors mail addresses. Following Abadi and Cardelli's calculus of Primitive Objects [AC94], actor's different behaviours are represented by mutually recursive records of methods only accessible by communication.

Cap does not follow all the principles of Agha's actors, but provides behaviours and addresses as primitives that allow to express very easily actor programs. We will assume some restrictions on Cap programs in order to define our analyses in a strict actor framework.

The remaining part of this section is devoted to a quick introduction to Cap syntax (a more precise presentation of the calculus and its semantic are available in [CPS96,CPS97]).

1.1 Cap Syntax

As a first example of a Cap expression, we construct a term corresponding to the "one-slot buffer" beginning with an empty state which is sent a *put* message.

$$\begin{aligned}&\nu a, b(a \rhd [\, put(v) = \zeta(e, s_e)(e \rhd [get(c) = \zeta(e', s_f)(c \lhd rep(v) \parallel e' \rhd s_e)])] \\ &\qquad \parallel a \lhd put(b))\end{aligned}$$

First, we create the two actor names a and b using the ν operator. An actor is built (via $\rhd$) by the association of an address (a) and a behaviour. In the previous example, the behaviour of a has two states recursively defined (via ζ). The first state (the *empty* buffer) only understands one *put* message and then switches to the second state (the *full* buffer) where it can understand only one *get* message. Before switching back to *empty*, it sends (via $\lhd$) the value coming from the corresponding *put* request to the argument of the *get* message.

When an actor accepts a message, $\zeta(e, s)$ binds the actor's *address* to e (called *ego*) and its *current behaviour* to s (called *self*). This operator is inspired by the ς defined by Abadi and Cardelli [AC94] to formalize self-substitution in objects. In our context, the capture of *self* and *ego* is used to formalize behaviour changes without introducing a fixpoint operator.

To define Cap syntax the following sets are used: Var an infinite set of variable symbols (x,x_i,e_i,$s_i \in Var$) and $Label$ a finite set of feature labels ($m_i \in Label$). Sequences of symbols are represented by a tilde ($\tilde{\ }$).

A configuration is a concurrent combination of actors and messages sent to actors. Their set $\mathcal{C}$ is built by the following grammar:

$$\begin{aligned}C &::= \ \emptyset \mid C \parallel C \mid \nu x.C \mid x \rhd T \mid x \lhd m(\tilde{T}) \mid (C) \\ T &::= \ [m_i(\tilde{x_i}) = \zeta(e_i, s_i)C_i]^{i \in I} \mid x \mid (T)\end{aligned}$$

The configurations C represent respectively: an empty process, processes in parallel, creation of actor's name, an actor named x waiting for a message (input transition) with T as its current behaviour and finally a message labelled m with its parameters $\tilde{T}$ sent to an actor x (output transition).

The terms T are either behaviours (between brackets) or variables. A behaviour is a set of reactions C_i to the labelled messages m_i (with their parameters $\tilde{x_i}$) that an actor can handle at a given time.

Syntactically, the sharing of the same address by several different actors is not forbidden. So, we will assume that Cap programs respect a linearity hypothesis: actors are not allowed to change the behaviour of other actors and only one behaviour is associated to an actor at each time (changing behaviour is only expressed using the *ego* variable: we will write $a \rhd [m_i() = \zeta(e_i, s_i)C \parallel e_i \rhd b_i]$ and not $a \rhd [m_i() = \zeta(e_i, s_i)C \parallel a \rhd b_i]$). Moreover, in this paper, values in parameters are only names of actors and not behaviours; this point will be discussed in conclusion.

2 Computation of the Interface of an Actor

The interface or behavioural type of an actor shall merely denote a finite-state transition system, where each state (respectively each transition) corresponds to an input-waiting state of the actor (respectively a label of an understood message at this current state). Thus we abstract away output events, as well as dynamic creation of actors, and even messages contents. Of course, as such, our analysis here is not intended to deal with properties of a complete actor (or process-calculi) term, but to focus on the specific input part of such a term. We insist on the fact that we don't provide in this paper any type discipline for a full actor language (as it is the main subject of some of our past papers [CPS97,CPDS99]).

A first part describes the translation of an actor term into a behavioural type. Then a finite-state transition system for these types is defined, as an operational semantics. We are indeed interested in the product system $Mailbox \times Control$, where $Mailbox$ represents the set of possible mailbox contents, i.e. multisets of labels and $Control$ is the first transition system. Then we give the operational semantics of this product system, which we shall further call an asynchronous transition system. Due to the unbounded nature of the mailbox, this system is an infinite-state system, but yet from any given initial state, only a finite number of steps can occur, until the mailbox is empty or we are stuck in a state where input capabilities don't match the mailbox content.

2.1 Behavioural Types

We adopt the TyCO syntax for behavioural types [RV00], i.e. types that denote the possible sequences of method invocations for a given actor. We recall here the grammar of behavioural types (without the "$\parallel$" operator and message parameters):

Definition 1 (Behavioural Types).

$$\alpha \;::=\; \Sigma_{i\in I}\mathtt{m}_i.\alpha_i \mid \Sigma_{i\in I}\tau.\alpha_i \mid \mu t.\alpha \mid t \mid \mathtt{0}$$

The labelled sum $\Sigma_{i\in I}\mathtt{m}_i.\alpha_i$ gathers together several method types to form the type of an actor that offers the corresponding set of methods. $\mathtt{0}$ denotes the sum with the empty indexing set.

The silent sum $\Sigma_{i\in I}\tau.\alpha_i$ is used to represent the internal non-determinism of an actor (a choice for instance). As soon as this choice is carried out, the actor behaves according to one of the types α_i.

2.2 Extraction of Behavioural Types

In this section, we present how to extract behavioural types from a Cap program. The principle is to keep only information related to the actors' input capacities. The type of each actor is produced and then analysed (represented by the function $\mathcal{O}$). We ignore the sending of messages and the parameters (as they are not behaviours).

As the creation of names and the installation of behaviours are separated, we have to carry a list of associations (variable of actor $\mapsto$ behavioural type) produced by the $x \rhd T$ configurations and then to extract the right behavioural type in the creation configurations $\nu x.C$. The list of associations is represented by ML lists (the operator "::" adds an element to a list and "@" merges two lists).

The configurations and the terms are typed in an environment Δ containing the *self* variables of each behaviour. The analysis of a term only returns one behavioural type, considering the linearity hypothesis we have made.

The following rules describe the process of extracting behavioural types and are explained thereafter:

$$\frac{}{\Delta \vdash \emptyset : \mathtt{0}} \qquad \frac{}{\Delta \vdash x \lhd m(\tilde{T}) : \mathtt{0}} \qquad \frac{\Delta \vdash C_1 : l_1 \quad \Delta \vdash C_1 : l_2}{\Delta \vdash C_1 \parallel C_2 : l_1 @ l2}$$

$$\frac{\Delta \vdash C : l \quad \mathcal{E}(x, l) = (\alpha, l') \quad \mathcal{O}(\alpha)}{\Delta \vdash \nu x.C : l'} \qquad \frac{\Delta \vdash T : \alpha}{\Delta \vdash x \rhd T : [x \mapsto \alpha]}$$

$$\frac{\Delta, s_i : \alpha \vdash C_i : l_i \quad \mathcal{E}(e_i, l_i) = (\alpha_i, [])}{[m_i(\tilde{x_i}) = \zeta(e_i, s_i)C_i]^{i \in I} : \mu\alpha.\Sigma_{i \in I}\mathtt{m}_i.\alpha_i} \qquad \frac{}{\Delta \vdash x : \Delta(x)}$$

$$\frac{\mathcal{E}(x, l) = (\alpha', l')}{\mathcal{E}(x, y \mapsto \alpha :: l) = (\alpha', y \mapsto \alpha :: l')} \qquad \frac{}{\mathcal{E}(x, x \mapsto \alpha :: l) = (\alpha, l)} \qquad \frac{}{\mathcal{E}(x, \emptyset) = (\mathtt{0}, \emptyset)}$$

The empty configuration as well as the sending of message have the null type $\mathtt{0}$. The parallel configuration gathers the associations from the two configurations.

The typing of the name creation consists in typing the configuration and then in extracting the behavioural type of this name, which is then analysed by the system described in the next sections (function $\mathcal{O}$). The resulting configuration returns the trailing associations. The installation of a behaviour produces a list with one element: the association of the variable x and the behavioural type resulting from the analysis of the term T (considering our hypothesis).

A reaction to a message is typed in an environment containing the *self* associated with the final type of the behaviour. The type of the next behaviour is extracted from the list of associations l_i which contains at least one element (the next behaviour is precised or not and an actor can't install a behaviour on another actor). This point is expressed by enforcing the outcome of the function $\mathcal{E}$ to be the empty list. Finally, the behavioural type is expressed by a fixpoint and gathers all the branches of the behaviour expression.

The type of a variable is given by the environment Δ if it represents effectively a behaviour (as we consider correct programs according to the first type system of [CPS97]).

The extraction function $\mathcal{E}(x, l)$ gives the behavioural type associated with x in the list l, if it exists, or the null type `0` instead. It also returns the tail of the list (without the association concerning x).

Example. According to this system, the one-slot buffer has the following simple type: $\mu\alpha.get.\mu\alpha'.put.\alpha$.

We can remark that we have not used the sum type with anonymous transitions. This type is required for languages allowing a kind of choice (for instance the conditional statement *if*).

2.3 Operational Semantics of Behavioural Types

We give a semantics of the types via a labelled transition relation where l denotes an element of *Label* $\cup \{\tau\}$. A label $\mathtt{m}_i$ - a basic transition - corresponds to the invocation of a method with name $\mathtt{m}_i$ whereas the label τ denotes a silent transition that corresponds to an internal hidden computation.

Our transition rules for the control part take the form $\Gamma \vdash p \xrightarrow{l} q$, where Γ is an environment of (recursive) behaviour definitions, p and q are behaviours (α-terms), and l is the label of the transition, $\mathtt{m}_i$ or τ. The term $\Gamma[t]$ denotes the definition p associated to the variable t in Γ, this association being denoted $t \mapsto p$.

Definition 2 (Transition rules for processes control part).

$$\frac{j \in I}{\Gamma \vdash \Sigma_{i \in I} l_i.\alpha_i \xrightarrow{l_j} \alpha_j} \quad \frac{\Gamma \vdash \Gamma[t] \xrightarrow{l} p}{\Gamma \vdash t \xrightarrow{l} p} \quad \frac{\Gamma \cup \{t \mapsto p\} \vdash t \xrightarrow{l} q}{\Gamma \vdash \mu t.p \xrightarrow{l} q}$$

2.4 Operational Semantics of Asynchronous Systems

We can now build the product system of control transitions and data transitions. the transition rules take the form $\Gamma \vdash (p, \omega) \xrightarrow{\mathtt{m}} (p', \omega')$ where ω (respectively ω') is the mailbox (i.e. a set of messages) relative to p (respectively to p'). Therefore τ-transitions have been discarded.

Definition 3 (Transition rules for asynchronous processes).

$$\frac{\Gamma \vdash p \xrightarrow{\mathtt{m}} q}{\Gamma \vdash (p, \omega \cup \{\mathtt{m}\}) \xrightarrow{\mathtt{m}} (q, \omega)} \quad \frac{\Gamma \vdash p \xrightarrow{\tau} q \qquad \Gamma \vdash (q, \omega) \xrightarrow{\mathtt{m}} (q', \omega')}{\Gamma \vdash (p, \omega) \xrightarrow{\mathtt{m}} (q', \omega')}$$

When not necessary, we shall merely ignore the environment part (Γ) of transition systems and consider only systems where Γ is abstracted away.

2.5 About Mailboxes

A mailbox, i.e. a multiset of labels, may be seen as a strictly positive integer vector (or equivalently as a function) in $\mathbb{N}^{Label}$, and we feel free to omit this coercion and to test membership of mailbox in constraint solutions when clear from the context. In the remaining, the main issue about these multisets will be to determine whether it is possible or not to define their possible values within Presburger arithmetics (additive integer arithmetics), i.e. as Presburger formulas about their components as integer vectors. The set *Label* always depends upon a given actor term under analysis, in which the number of different labels is obviously finite and known.

3 Expressing Input Capabilities of Asynchronous Systems

The type we extract from an actor is not the end of the road, as we must now work out the possible meanings of *message-not-understood* like errors, with in mind the ability to devise type systems from these definitions.

The dedicated logic we propose to explore these issues is a very expressive mix of CTL-like temporal features (see [CES86]) to specify future behaviours and Presburger arithmetics (see [Pug91]) to specify mailbox contents, well suited to expressing pervasive and important properties relative to existence of reduction paths and emptiness of mailbox. We first define the grammar $\mathcal{F}$ of such properties.

3.1 Logic for Asynchronous Systems (LAS)

Definition 4 (The temporal logic LAS).

$$\begin{array}{rcll} \mathcal{F} & ::= & \exists\Diamond\ \mathcal{F} \mid \exists\bigcirc\ \mathcal{F} & \textit{(temporal operators)} \\ & & \mid \neg\mathcal{F} \mid \mathcal{F} \vee \mathcal{F} \mid \textit{true} & \textit{(boolean operators)} \\ & & \mid \exists\Uparrow\ \mathcal{F} \mid \exists x.\mathcal{F} \mid \mathcal{E} \geq \mathcal{E} & \textit{(arithmetical operators)} \end{array}$$

where $\mathcal{E}$ denotes an expression belonging in Presburger arithmetics. As usual, the other standard relational operators ($=, \neq, >, \leq$ and $<$) can easily be retrieved from boolean operations and $\geq$.

Variables occurring in arithmetical expressions are either fresh quantified integer variables (introduced with $\exists x \ldots$) or label variables denoting the number of corresponding messages (with the same label) in the mailbox. For a constraint $e_1 \geq e_2$, the term $[\![e_1 \geq e_2]\!]$ shall denote its set of integer solutions.

Then, we give the satisfaction relation of our logic with respect to the underlying transition system.

Definition 5 (Satisfaction relation for LAS). *The satisfaction relation between a configuration (p, ω) and a formula f, denoted $(p, \omega) \vDash f$, is the smallest fixpoint of the following structural inductive rules on f:*

$$\frac{\exists \omega' :\ (p, \omega \cup \omega') \vDash f}{(p,\omega) \vDash \exists\Uparrow f} \qquad \frac{\exists (p', \omega') :\ \vdash (p,\omega) \longrightarrow^{*} (p',\omega') : (p',\omega') \vDash f}{(p,\omega) \vDash \exists\Diamond\ f}$$

$$\frac{(p,\omega) \nvDash f}{(p,\omega) \vDash \neg f} \qquad \frac{\exists (p', \omega') :\ \vdash (p,\omega) \longrightarrow (p',\omega') : (p',\omega') \vDash f}{(p,\omega) \vDash \exists\bigcirc\ f}$$

$$\frac{(p,\omega) \vDash f[K/x]}{(p,\omega) \vDash \exists x.f} \qquad \frac{(p,\omega) \vDash f_1}{(p,\omega) \vDash f_1 \vee f_2} \qquad \frac{(p,\omega) \vDash f_2}{(p,\omega) \vDash f_1 \vee f_2}$$

$$\frac{\omega \in [\![e_1 \geq e_2]\!]}{(p,\omega) \vDash e_1 \geq e_2} \qquad \frac{}{(p,\omega) \vDash \mathit{true}}$$

As usual, we shall sometimes identify a formula f with the set of configurations where f holds, i.e. its denotation $[\![f]\!] \triangleq \lambda p.\{\omega \mid (p,\omega) \vDash f\}$

From these operators, we can further define the following abbreviations for practical purposes:

Definition 6 (Derived operators of LAS).

`blocked`	$\triangleq \neg\exists\bigcirc\ \mathit{true}$	`empty_mb`	$\triangleq \bigwedge_{\mathtt{m}\in Label}(\mathtt{m} = 0)$
$f_1 \wedge f_2$	$\triangleq \neg(\neg f_1 \vee \neg f_2)$	$f_1 \Rightarrow f_2$	$\triangleq \neg f_1 \vee f_2$
$\forall\Uparrow f$	$\triangleq \neg\exists\Uparrow \neg f$	$\forall x.f$	$\triangleq \neg\exists x.\neg f$
$\forall\Box\ f$	$\triangleq \neg\exists\Diamond\ \neg f$		

Moreover, we shall say that a formula f is "downward-closed", whenever formula $\exists\Uparrow f \Rightarrow f$ holds.

4 Decidability of Model-Checking in LAS

We state here very succinctly that for any formula $f \in$ LAS and any behavioural type p, $[\![f]\!](p)$ is Presburger definable, i.e. the set of label multisets (or mailbox contents) satisfying $[\![f]\!](p)$ is definable as a formula in Presburger arithmetics. It follows that model-checking LAS formulas with respect to behavioural types is decidable because model-checking in Presburger arithmetics is decidable too.

This section is organised as follows. First, we recall Presburger definability of flattened regular languages. Then, we handle the LAS operators.

4.1 Flattening Regular Languages

Definition 7 (Multiset interpretation of words). *For a given word w (i.e. a finite string of letters), we note $w^\flat$ its "flat" interpretation as the multiset of its letters, whatever their original order in w. Also, we define its pointwise extension to sets of words (i.e. languages).*

Now we state that for any regular language W, $W^\flat$ is a semilinear set, and as such is Presburger definable.

Theorem 1 ($W^\flat$ belongs in Presburger arithmetics). *For any regular language W, $W^\flat$ is a semilinear set, i.e.*

$$W^\flat \triangleq \bigcup_{i\in I}\{v_i + M_i.\kappa \mid \quad v_i \in \mathbb{N}^{Label} \wedge M_i \in \mathbb{N}^{Label\times d_i} \wedge \kappa \in \mathbb{N}^{d_i}\}$$

(where d_i, v_i and M_i are non-trivially derived from W) and is therefore Presburger definable.

Proof (reference). This result is known since the original works of Parikh, see [Esp95] for a complete insight.

4.2 Example

To get the reader used to flattening regular languages, we illustrate this operation on a small artificial example. Starting from:

$$W = \mathtt{a}.(\mathtt{b}.(\mathtt{c}+\mathtt{d}))^* + \mathtt{e}$$

we get:

$$W^\flat = (\mathtt{a}.(\mathtt{b}.(\mathtt{c}+\mathtt{d}))^* + \mathtt{e})^\flat = (\mathtt{a}.(\mathtt{b}.\mathtt{c})^*.(\mathtt{b}.\mathtt{d})^* + \mathtt{e})^\flat$$

from which, applying theorem 1, we finally get $I = \{1,2\}$, $d_1 = 2$, $d_2 = 0$ and:

$$\begin{array}{l|l} v_1 = (1,0,0,0,0) & v_2 = (0,0,0,0,1) \\ M_1 = \begin{pmatrix} 0,1,1,0,0 \\ 0,1,0,1,0 \end{pmatrix} & M_2 = () \end{array}$$

or equivalently, using Presburger arithmetics:

$$(\mathtt{a} = 1 \wedge \mathtt{b} = \mathtt{c}+\mathtt{d} \geq 0 \wedge \mathtt{e} = 0) \;\vee\; (\mathtt{a} = \mathtt{b} = \mathtt{c} = \mathtt{d} = 0 \wedge \mathtt{e} = 1)$$

4.3 Computing Denotation for Full LAS

Theorem 2 (LAS belongs in Presburger arithmetics). *Given a LAS formula f and a behavioural type p, $[\![f]\!](p)$ is Presburger definable.*

Proof (reference). A proof for similar concerns (namely CTL logic for Basic Parallel Processes) can be found in [Esp97]. In particular, the proof for the $\exists\Diamond$ operator uses theorem 1. Adapting the proof to our systems and logic is straightforward.

5 Expressivity of LAS: About Orphan Messages

Here, with the help of our expressive logic LAS, we explore different properties whose common goal is to avoid communication errors.

A pervasive feature will be the downward-closedness of these properties, which ensures that we can use them in type systems. Indeed, the same way that a behavioural type is merely an abstraction of real input behaviours, the set of messages that will effectively be received by any actor cannot be computed very precisely in the general case. In order to devise a type system, we need to compute an upper approximation of incoming messages, which we somehow match against input capabilities (see [CPS97,CPDS99]). So, in the end, as an actor will receive no more messages than computed, and usually strictly less, a simple solution is to use a downward-closed input language, so that we avoid many spurious typing errors.

The different properties exposed in this section will be illustrated by the one-slot buffer example and a more complex one $\mu q_0.\mathtt{a}.(\mathtt{d} + \mathtt{b}.\mu q_2.(\mathtt{a}.q_2 + \mathtt{c}).q_0)$ pictured as a finite state transition system in the following figure 1.

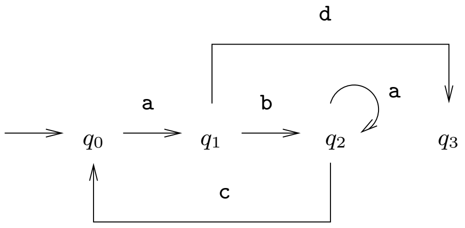


Fig. 1. A small example.

This example could represent a classic application with a main control loop with a little inner loop (with message `a`) and an exit branch (with message `d`).

We give results, i.e. $[\![f]\!]$ where f is the property under scrutiny, only for the initial state of our example systems, and as flattened regular expressions (except for the first example).

5.1 Strongly Safe Input Language

A very natural idea that comes first to mind is to rule out any configuration that may lead to a blocked situation, where mailbox is not empty and yet no further reduction is ever possible. A simple formula suffices to express this strong requirement:

$$\mathit{Strong} \triangleq \forall\Box\ (\mathtt{blocked} \Rightarrow \mathtt{empty\_mb}) \equiv \forall\Diamond\mathtt{empty\_mb}$$

This property entails a very fine-grained analysis of input capabilities, and therefore is very valuable for an end-user. We may slightly decrease its acuteness, by taking its downward closure $\exists\Uparrow \mathit{Strong}$.

Examples. For the one-slot buffer, we obtain for $[\![\mathit{Strong}]\!](\alpha)$:

$$\mathtt{get} \geq 0 \land \mathtt{get} \geq \mathtt{put} - 1 \land \mathtt{put} \geq \mathtt{get}$$

which can be represented by the flattened regular expression:

$$((\mathtt{put}.\mathtt{get})^* + \mathtt{put}.(\mathtt{put}.\mathtt{get})^*)^\flat$$

For example of figure 1, we obtain for $[\![Strong]\!](q_0)$:

$$(\Lambda + \mathtt{a} + \mathtt{a.b} + \mathtt{a}^2.\mathtt{b.a}^* + \mathtt{a.b.c} + \mathtt{a}^2.\mathtt{b.c} + \mathtt{a.d})^\flat$$

First, whether the system will take the exit branch or stay within the main loop depends on the presence of a message d.

Second, assuming we are stuck in the main loop, the inner loop adds a major constraint because we can decide to handle message a either in state q_0 or in state q_2. So we can send at most two messages a with b and c, because otherwise we could get stuck in state q_0 with b and/or c orphan messages.

Without the inner loop, the result becomes:

$$(\mathtt{a} + \mathtt{a.d} + \mathtt{a.(a.b.c)}^+ + \mathtt{(a.b.c)}^* + \mathtt{a.b.(a.b.c)}^*)^\flat$$

expressing that the actor may loop the main loop any number of times (except if message d is sent).

Discussion. As a conclusion, this property is indeed too strong as for instance it doesn't allow interferences between nested loops, as illustrated by our example 1. This stems from the fact that our formula requires that every message should be handled in every computation path from the initial state of an actor, whereas in real life, input/output communication patterns may ensure a strict sequence of messages, that will be received one at a time. So, as a conclusion, the *Strong* property is not really suited to build a loose enough type system, one that would allow real life applications to be well-typed.

Moreover, a type system would enjoy better properties if well-typedness of a given system would imply well-typedness of each of its sub-process taken in isolation. For instance, the correctness of a process should not rely upon its environment sending a message that unblocks the treatment of otherwise orphan messages already present in an actor's mailbox.

The following alternative definitions of input languages we shall retain for designing type systems does reflect this simulation of an angelic environment that provides unblocking messages when needed. They are also downward-closed sets of messages, i.e. every subset of a solution is a solution too.

5.2 First Orphan-Free Input Language

This input language is only a rephrasing in our logic of earlier works presented for instance in [CPDS99]. As this language suffers from some important weaknesses, it is mentioned here mainly for historical reasons, and in witness of our claim for the expressivity of our logic.

$$First(\mathtt{m}) \triangleq \forall m_0 . m_0 = \mathtt{m} \Rightarrow \forall\Uparrow \forall d . d = \mathtt{m} - m_0 \Rightarrow \forall\Box\, \exists\Uparrow \exists\Diamond\, \mathtt{m} \leq d$$

Sketchily, this property ensures that an actor will have at any time the ability to treat any message m in its mailbox, provided that a kind environment may send unblocking messages during the execution. For finite branches, this property boils down to computing multiset intersection of such paths. The full input language is then the intersection of $First(\mathtt{m})$ for every label m.

Examples. The result for the one-slot buffer is not a surprise, because of its linear behaviour: $(\texttt{get}^*.\texttt{put}^*)^\flat$. For example of figure 1, we find: $(\Lambda + \texttt{a} + \texttt{d} + \texttt{a.d})^\flat$ that doesn't allow to loop the main loop even if no message d is ever sent.

Discussion. Our present formulation doesn't do justice to the simplicity of the original framework, whose computation only involved a simple greatest fixpoint, instead of a rather complicated formula. Yet, our formula may be computed with respect to every label independently, which makes it worthwhile for complexity and efficiency issues. This property has already been used in a type system for the full CAP actor's language, despite the fact that finite branches dominate the computation, no matter how many loops may exist. In practice, this drawback totally prevents programmers from using exit branches, so that only actors with a finite or a forever cycling lifetime may get well typed. Moreover, finite branches with no common message (for instance, a choice between two different messages) are systematically ill-typed, because the intersection of all finite paths is empty in this case.

5.3 Second Orphan-Free Input Language

This last input language is currently the best answer as it fulfills our requirements. It involves a slight extension of our logic, because we need to compute a greatest fixpoint. This fixpoint obviously needs to be well defined and computed within a finite amount of time.

Definition 8 (Greatest fixpoint in LAS). *For any LAS property scheme $F(X)$ where X is a property variable, then we define $\nu X.F(X)$, provided the following constraints hold:*

1. *X only occurs nested inside an even number of "$\neg$" operators. $F(X)$ is then monotonous, and least as well as greatest fixpoints are then well defined.*
2. *$F(X)$ is downward-closed. Computation of greatest fixpoint (starting from $X_0 \triangleq true$) then terminates, because there is no infinite strictly decreasing chains in $\mathbb{N}^{Label}$.*

Theorem 3 ($\nu X.F(X)$ is Presburger definable). *Any term $\nu X.F(X)$ satisfying the previous conditions is Presburger definable.*

Proof (sketch). During fixpoint computation, the first iterate (*true*) is obviously Presburger definable, and each new iterate is itself Presburger definable from the previous one. Termination is obtained by deciding implication between two successive iterates.

Now, we define our second language. We base our definition on three simple facts. First, an empty mailbox should always be accepted. Second, adding some unblocking messages to an accepted configuration should lead to another accepted configuration, whatever the execution path. Third, added messages should not be orphan, i.e. one execution path that can handle all added messages should exist:

$$Second \triangleq \nu X.\exists\Uparrow (\exists\Diamond\ \texttt{empty\_mb} \wedge \forall\Box\ X)$$

Examples. The one-slot buffer has the same result: $(\texttt{get}^*.\texttt{put}^*)^\flat$. For example of figure 1, we get: $(\texttt{d} + \texttt{a.d} + \texttt{a}^*.\texttt{b}^*.\texttt{c}^*)^\flat$ so, either the exit branch is followed or the actor accepts any number of messages a,b and c. Without the inner loop, the result is still the same as we doesn't take deadlocks into account, indeed in this case our angelic environment can always safely solve such deadlocking configurations, i.e. without sending some other orphan message.

Discussion. This property has a very simple expression, and this time we can cope with finite branches and loops without losing precision, or computing a severe under-approximation. This is the right choice, as shown in the next section, under the assumption that fixpoint computations will never be too much resource demanding to be a part of an intensively used type system.

6 Orphan-Free Input Language and Full Type Discipline

In order to embed our input languages in a full type discipline, we should state some continuity and subject-reduction properties of the type system. Unfortunately, we lack room to put forward our ideas about typing actors and invite the interested reader to consult [CPS97,CPDS99].

Nevertheless, we can state invariance properties which should help the reader to convince himself about the possibility of using our *Second* input language to state subject-reduction properties dedicated to ruling out communication errors in real executions of actor configurations.

The following theorem handles blocked as well as unblocked configurations, expressed at the level of behavioural types. The first case represents one execution step, whereas the second one deals with blocked configurations. As shown in our earlier works, systems that may get stuck in a blocked configuration may still get well-typed, under our assumption of an angelic environment.

Theorem 4 **(*Second* is invariant).** *For any configuration (p, ω), message label* m *and p', we have:*

$$\frac{p \xrightarrow{\texttt{m}} p' \; (p, \omega \cup \{\texttt{m}\}) \models \mathit{Second} \wedge \neg\texttt{blocked}}{(p', \omega) \models \mathit{Second}} \qquad \frac{(p, \omega) \models \mathit{Second} \wedge \texttt{blocked}}{(p, \omega) \models \exists\Uparrow (\mathit{Second} \wedge \neg\texttt{blocked})}$$

Proof (omitted).

Our last theorem states the absence of a kind of communication errors called *static orphan* messages, namely messages which, once in an actor's mailbox, would never get a chance to be treated, whatever the (angelic) environment put in parallel.

Theorem 5 **(*Second* implies orphan-free).** *For any configuration (p, ω), we have:*

$$\frac{(p, \omega) \models \mathit{Second}}{(p, \omega) \models \exists\Uparrow \exists\Diamond \; \texttt{empty\_mb}}$$

Proof (omitted).

Conclusion

As a conclusion, we have achieved to devise an interesting general purpose logic to express behavioural properties, for some restricted kind of infinite-state systems. Our logic is expressive enough to rephrase many past and current studies about behaviours of actor programs. For the time being, we only deal with input capabilities, but it seems that it can be extended to handle transducers, i.e. input/output devices. Moreover, modeling message contents, as long as we only allow them to denote behaviours as well as addresses, is planned for future work. The main issue in this case will be to deal with the possibility for behaviours to send messages to global addresses, leading to rather intricate scope extrusion problems. As far as we know, our logic is quite unique and we haven't found yet any tool dealing with a similar logic of the same behavioural kind. Therefore we decided to implement a prototype model-checker for LAS, on top of a decision procedure for Presburger arithmetics [Pug91]. In a rather user-friendly fashion, our prototype logs every calculation it makes, as a sequence of literally written Presburger equations, so that the user can trace the computational meaning of its specification, if needed. The generated log file can serve as a basis for step-by-step debugging purposes for instance.

From a theoretical viewpoint, our work on LAS can be extended to handle full TyCo types, including the "$\|$" operator, as it is partially shown in [Esp97]. We can therefore apply our present work (including our full type discipline not presented here) to other concurrent programming languages, such as Pict [PT97].

References

[AC94] M. Abadi and L. Cardelli. A theory of primitive objects: untyped and first-order systems. In *Proc. Theor. Aspects of Computer Software*, 1994.

[Agh86] Gul Agha. *Actors: A model of concurrent computation in distributed systems.* MIT Press, Cambridge, Mass., 1986.

[Bou97] Gérard Boudol. Typing the use of resources in a concurrent calculus. In R. K. Shyamasundar and K. Ueda, editors, *Proceedings of ASIAN '97*, volume 1345 of *LNCS*, pages 239–253. Springer-Verlag, 1997.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CPDS99] Jean-Louis Colaço, Marc Pantel, Fabien Dagnat, and Patrick Sallé. Static safety analysis for non-uniform service availability in actors. In *Proc. of FMOODS*, February 1999.

[CPS96] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. CAP: An actor dedicated process calculus. In *Prof. of Proof Theory of Concurrent Object-Oriented Programming*, May 1996.

[CPS97] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. A set-constraint-based analysis of actors. In *Proc. of FMOODS*, July 1997.

[CRR02] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models: model checking message-passing programs. In *Symposium on Principles of Programming Languages*, pages 45–57, 2002.

[DPCS00] Fabien Dagnat, Marc Pantel, Matthias Colin, and Patrick Sallé. Typing concurrent objects and actors. *L'Objet – Méthodes formelles pour les objets*, Volume 6(1/2000):pages 83–106, May 2000.

[Esp95] Javier Esparza. Petri nets, commutative context-free grammars, and basic parallel processes, 1995.

[Esp97] Javier Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2):85–107, 1997.

[FLMR97] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implicit typing à la ml for the join-calculus. In *Proc. of CONCUR*, LNCS 1283, pages 196–212. Springer-Verlag, 1997.

[HBS73] Carl Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proc. of Int. Joint Conference on Artificial Intelligence*, 1973.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proc. ECOOP '91*, July 1991.

[Kob97] Naoki Kobayashi. A partially deadloack-free typed process calculus. In *Proc. of the conf. Logic In Computer Science*, 1997.

[Mil91] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, LFCS, 1991.

[Nie95] Oscar Nierstrasz. Regular types for active objects. In *In Object-Oriented Software Composition, ACM SIGPLAN Notices*, pages 99–121. Prentice Hall, October 1995.

[NNS99] E. Najm, A. Nimour, and J-B. Stefani. Infinite types for distributed object interfaces. In *Proc. of FMOODS*, February 1999.

[PT97] B. Pierce and D. Turner. Pict: A programming languages based on the π-calculus. Technical Report 476, Indiana Univ., March 1997.

[Pug91] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, pages 4–13, 1991.

[Pun96] Franz Puntigam. Type for active objects based on trace semantics. In *Proc. of FMOODS*, pages 5–20, March 1996.

[RR01] Sriram K. Rajamani and Jakob Rehof. A behavioral module system for the pi-calculus. *Lecture Notes in Computer Science*, 2126:375–??, 2001.

[RV00] António Ravara and Vasco T. Vasconcelos. Typing non-uniform concurrent objects. In *CONCUR'00*, volume 1877 of *LNCS*, pages 474–488. Springer-Verlag, 2000.

[Wal95] D. Walker. Objects in the π-calculus. *Information and Computation*, (116), 1995.

The Kell Calculus: Operational Semantics and Type System

Philippe Bidinger and Jean-Bernard Stefani

INRIA Rhône-Alpes, 38334 St Ismier, France
{philippe.bidinger,jean-bernard.stefani}@inrialpes.fr

Abstract. This paper[1] presents the Kell calculus, a new distributed process calculus that retains the original insights of the Seal calculus (local actions, process replication) and of the M-calculus (higher-order processes and programmable membranes), although in a much simpler setting than the latter. The calculus is equipped with a type system that enforces a unicity property for location names that is crucial for the efficient implementation of the calculus.

1 Introduction

Numerous distributed process calculi have been introduced in the past ten years. One of the calculi that has received the most attention has been Mobile Ambients [5], as witnessed by the numerous variants that have been proposed to overcome some of its perceived deficiencies: Safe Ambients (SA) [11], Safe Ambients with passwords [12], Boxed Ambients (BA) [3], Controlled Ambients (CA) [16], New Boxed Ambients (NBA) [4], Ambients with process migration ($\mathbf{M}^3$) [7].

Mobile Ambients are unfortunately costly to implement in a distributed setting (i.e. with ambients representing potentially widely separated sites), in particular because of the synchronization implied in its migration primitives. Consider the reduction rule associated with the *in* primitive of Mobile Ambients:

$$n[\mathrm{in}\, m.P \mid Q] \mid m[R] \rightarrow m[R \mid n[P \mid Q]]$$

This rule mandates a rendez-vous between ambient n and ambient m. Thus, if ambient n and ambient m are taken to represent two remote sites, a faithful implementation of this rule would require some form of distributed synchronization.

The difficulty of implementing Mobile Ambients in a distributed setting and the need for two and even three-way-synchronization between ambients to implement Ambient migration primitives, has been made clear by two implementation attempts. The first one, reported in [9], implements the original Mobile Ambients calculus using (an implementation of) the Distributed Join calculus. The second one, reported in [13], describes a Safe Ambients abstract machine, called PAN, that alleviates some of the difficulty inherent in Mobile Ambients implementation by implementing a variant of the original calculus with co-capabilities and single-threadedness [11], but where ambients no longer correspond to physical loci of computations.

[1] This work has been supported in part by the Mikado project – IST-2001-32222.

Recent variants of Ambients, such as Boxed Ambients (BA) and New Boxed Ambients (NBA) propose a model which combines local communication across location boundaries (inspired by the Seal calculus [17]), and the Ambient migration primitives `in` and `out`. In a model such as NBA, communication can be implemented efficiently while migration primitives still imply in general a distributed rendez-vous. This is much preferrable to the original Mobile Ambients, but still raises a number of questions.

First, one can think of turning the Ambient migration primitives into asynchronous ones. This would be useful to take into account the possibility of failure for migration, especially in wide-area settings. To illustrate, one could think of splitting the Mobile Ambients `in` primitive into a pair of primitives `move` and `enter` whose behavior would be given by the following reduction rules (we use co-capabilities and passwords, as in the NBA calculus):

$$n[\mathtt{move}\langle m,h\rangle.P \mid Q] \mid \overline{\mathtt{move}}(x,y).R \rightarrow \mathtt{enter}\langle n,m,h,P,Q\rangle \mid R\{m/x,h/y\}$$

$$\mathtt{enter}\langle n,m,h,P,Q\rangle \mid m[\overline{\mathtt{enter}}(x,h).S \mid T] \rightarrow m[S\{x/n\} \mid T \mid n[P \mid Q]]$$

In so doing, note that migration primitives now look very much like higher-order communication across location boundaries. Second, one may envisage further extensions allowing more sophisticated authentication schemes, or dynamic security checks (e.g. additional parameters for proof-carrying code schemes). This in turn would further strengthen the similarity between migration primitives and higher-order communication. Third, there are still pending questions concerning migration primitives and their combination. For instance, should we go for communications à la Boxed Ambients or should we consider instead to split up the migration primitives such as `to` migration primitive in the $\mathbf{M}^3$ calculus, yielding a form of communication similar to Dπ [10] or Nomadic Pict [18], where communication is a side-effect of process migration ? Should we allow for more objective forms of migration to reflect control that ambients can exercize on their content ?

Our answer to these questions is to move away from the Ambient primitives altogether, and instead to follow the lead of higher-order process calculi such as $D\lambda\pi$ [19] and the M-calculus [14], where process migration is a side-effect of higher-order communication. Indeed, as demonstrated in the M-calculus, higher-order communication, coupled with programmable localities, provides the means to model different forms of migration protocols, and different forms of locality semantics. The M-calculus avoids embedding predefined choices concerning migration primitives and their interplay. Instead, these choices can be defined, within the calculus itself, by programming the appropriate behavior in locality "membranes" (the control part P of an M-calculus locality $a(P)[Q]$). The M-calculus, however, may appear as rather complex, especially with respect to Mobile Ambients. The reduction relation of the calculus, which defines its operational semantics, contains several so-called routing rules that govern the crossing of location boundaries. Clearly it would be interesting to explain these different rules as instances of basic primitive "boundary crossing" cases.

The calculus we introduce in this paper is an attempt to define a calculus with process migration and hierarchical localities, that avoids the need for distributed synchronization, while preserving the simplicity of Mobile Ambients, and retaining the basic insights of the M-calculus: migration as higher-order communication, programmable membranes for localities. We call this new calculus the Kell calculus (the word “kell” is a variation on the word “cell”, and denotes a locality or locus of computation).

Avoiding primitives with implied distributed synchronization is not the only requirement for an efficient implementation in a distributed setting. Today’s wide-area communication is predicated upon the existence of globally unique identifiers and addresses (e.g. IP addresses). It is therefore important, in a calculus intended as a foundation for wide-area distributed programming, to be able to enforce such a constraint, both for modelling and implementation purposes. In the Kell calculus, the unicity of addresses translates into the unicity of locality (kell) names. We show in the paper how to enforce the constraint by means of a polymorphic type system, inspired by the type system defined for the M-calculus.

The Kell calculus and its reduction semantics has already been introduced in [15], together with faithful encodings of Mobile Ambients and of the Distributed Join calculus. We present in this paper a variant of the Kell calculus with join patterns which is a more natural fit for the type system, together with a new reduction semantics.

The paper is organized as follows. Section 2 informally introduces the main constructs of the Kell calculus, together with several examples. Section 3 gives the syntax and operational semantics of the calculus. Section 4 defines a type system for the Kell calculus that enforces the unicity of kell name property. Section 5 concludes the paper with a discussion of related work and of directions for future research.

2 Introducing the Kell Calculus

The Kell calculus is in fact a family of calculi that share the same constructs and that differ only in the language of message patterns used in triggers (see below). In this section, we present informally the different constructs of the Kell calculus variant we use in this paper.

The core of the calculus is the asynchronous higher-order π-calculus. Among the basic constructs of the calculus we thus find:

- the *null* process, $\mathbf{0}$; names a, x, which also play the roles of (name and process) variables;
- the *restriction*, $\nu a.P$, where a is a name, P is an arbitrary Kell calculus process, and ν is a binding operator;
- the *parallel* composition, $P \mid Q$;
- *messages* of the form, $a\langle\widetilde{w}\rangle$, where a is a name, and where $\widetilde{w}$ is a (possibly empty) vector of elements w that can be either names or processes.
- *triggers*, or receivers, of the form $\xi \triangleright P$, where ξ is a *receipt pattern* and P is an arbitrary kell calculus process.

The patterns used in this paper correspond to an extension of the Join patterns, i.e. patterns of messages used in the Join calculus:

$$\xi ::= J \quad | \quad J \mid a[x] \qquad J ::= a\langle\widetilde{u}\rangle \quad | \quad a\langle\widetilde{u}\rangle^{\uparrow} \quad | \quad a\langle\widetilde{u}\rangle^{\downarrow} \quad | \quad J \mid J$$

where $\widetilde{u}$ is a vector of elements u. Each element u can be either a (bound) variable x, or a free name, which we note (x). Variables are bound in patterns and their scope extends to the process of the right-hand side of the trigger sign $\triangleright$. Free names (x) are not bound in the pattern.

To this higher-order π-calculus core, we add just one construct, the kell construct, $a[P]$, which is used to localize the execution of a process P at location (we say "kell") a.

In the Kell calculus, computing actions can take four simple forms, illustrated below:

1. Receipt of a local message, as in the reduction below, where a message, $a\langle Q\rangle$, on port a, bearing the process Q, is received by the trigger $a\langle x\rangle \triangleright P$ (notice that triggers, as in the Join calculus, are replicated, i.e. they persist after a reaction):
$$a\langle Q\rangle \mid (a\langle x\rangle \triangleright P) \;\rightarrow\; (a\langle x\rangle \triangleright P) \mid P\{Q/x\}$$
2. Receipt of a message originated from the environment of a kell, as in the reduction below, where a message, $a\langle Q\rangle$, on port a, bearing the process Q, is received by the trigger $a\langle x\rangle \triangleright P$, located in kell b (the pattern $a\langle x\rangle^{\uparrow}$ indicates that a message is expected from *outside* the local kell):
$$a\langle Q\rangle \mid b[a\langle x\rangle^{\uparrow} \triangleright P] \;\rightarrow\; b[(a\langle x\rangle^{\uparrow} \triangleright P) \mid P\{Q/x\}]$$
3. Receipt of a message originated from a sub-kell, as in the reduction below, where a message, $a\langle Q\rangle$, on port a, bearing the process Q, and coming from sub-kell b, is received by the trigger $a\langle x\rangle \triangleright P$, located in the parent kell of kell b (the pattern $a\langle x\rangle^{\downarrow}$ indicates that a message is expected from a kell *inside* the local kell):
$$(a\langle x\rangle^{\downarrow} \triangleright P) \mid b[a\langle Q\rangle \mid R] \;\rightarrow\; (a\langle x\rangle^{\downarrow} \triangleright P) \mid P\{Q/x\} \mid b[R]$$
4. Suspension of a kell, as in the reduction below, where the sub-kell named a is destroyed, and the process Q it contains is sent in a message on port b:
$$a[Q] \mid (a[x] \triangleright b\langle x\rangle) \;\rightarrow\; (a[x] \triangleright b\langle x\rangle) \mid b\langle Q\rangle$$

Actions of the form 1 above are standard π-calculus actions. Actions of the form 2 and 3 are just extensions of the message receipt action of the π-calculus to the case of triggers located inside a kell. They can be compared to the communication actions in Boxed Ambients or in the Seal calculus [6].

Actions in the Kell calculus obey a locality principle that states that any computing action should involve only one locality at a time (and its environment, when considering crossing locality boundaries). In particular, notice that there are no reductions in the calculus that, similar to the Mobile Ambients `in` move, would involve two adjacent kells. In particular, we *do not* have reductions of the following form:

$$a[\mathtt{in}\langle Q\rangle] \mid b[\mathtt{in}\langle x\rangle \triangleright x] \;\rightarrow\; a[\mathbf{0}] \mid b[(\mathtt{in}\langle x\rangle \triangleright x) \mid Q]$$

Actions of the form 4 are characteristic of the Kell calculus. They allow the environment of a kell to exercize control over the execution of the process located inside a kell. They can be compared to the migrate and replicate construct of the Seal calculus, but note that they provide more control over the execution of processes. Consider for instance the processes P and R defined as:

$$P \triangleq \mathtt{suspend}\langle(a)\rangle \mid a[x] \triangleright a\langle x\rangle \qquad R \triangleq \mathtt{resume}\langle(a)\rangle \mid a\langle x\rangle \triangleright a[x]$$

We have the following reductions:

$$\begin{aligned}\texttt{resume}\langle a\rangle \mid \texttt{suspend}\langle a\rangle \mid P \mid R \mid a[Q] &\to \texttt{resume}\langle a\rangle \mid P \mid R \mid a\langle Q\rangle \\ &\to P \mid R \mid a[Q]\end{aligned}$$

In this example, the environment of kell a first suspends its execution (there is no evaluation under a $a\langle . \rangle$ context), and then resumes it (processes can execute under a $a[.]$ context).

The higher-order nature of the calculus, together with the above control capability, allows the definition of different forms of programmable "membranes" around kells. For instance, a membrane around $a[K]$ can take the form: $c[M(a) \mid a[K]]$, in which case its behavior is defined by the process $M(a)$. Here are some simple examples of membranes (we assume that all messages *to* kell a have the form $\texttt{rcv}\langle a, op, args\rangle$ and that all messages *from* kell a have the form $\texttt{snd}\langle b, op, args\rangle$):

Transparent Membrane. This is a membrane that does nothing (it just allows messages destined to, or emitted by, a to be transmitted without any control):

$$M_{trans} \triangleq (\texttt{rcv}\langle (a), b, x\rangle^{\uparrow} \triangleright \texttt{rcv}\langle a, b, x\rangle) \mid (\texttt{snd}\langle b, c, x\rangle^{\downarrow} \triangleright \texttt{snd}\langle b, c, x\rangle)$$

Intercepting Membrane. This is a membrane that triggers behaviour $P(x)$ when a message $a\langle x\rangle$ seeks to enter kell a, and behaviour $Q(b, y)$ when a message $\texttt{m}\langle b, y\rangle$ seeks to leave kell a. Notice how this allows the definition of wrappers with pre and post-handling of messages:

$$M_{int} \triangleq (\texttt{rcv}\langle (a), b, x\rangle^{\uparrow} \triangleright \texttt{Pre}\langle b, x\rangle) \mid (\texttt{snd}\langle b, c, x\rangle^{\downarrow} \triangleright \texttt{Post}\langle b, c, x\rangle)$$

Migration Membrane. This is a membrane that allows new processes to enter kell a via the `enter` operation, and allows kell a to move to a different kell b via the `go` operation. Compare these operations with the asynchronous Ambient migration primitives `enter` and `move` given in Section 1, and the `go` primitive of the Distributed Join calculus:

$$\begin{aligned}M_{mig} \triangleq M_{trans} &\mid (\texttt{rcv}\langle (a), (\texttt{enter}), x\rangle^{\uparrow} \triangleright (a[y] \triangleright a[x \mid y])) \\ &\mid (\texttt{go}\langle b\rangle^{\downarrow} \triangleright (a[y] \triangleright \texttt{snd}\langle b, \texttt{enter}, a[y]\rangle))\end{aligned}$$

Membrane with Fail-Stop Failures. This is a membrane that allows to `stop` the execution of locality a (simulating a failure in a fail-stop model), and that implements a simple failure detector via the `ping` operation. Compare these operations with the π_{1l}-calculus [1], or the Distributed Join calculus, model of failures:

$$\begin{aligned}M_{fails} \triangleq \nu c\, f.M_{trans} \mid c &\mid (\texttt{stop}\langle (a)\rangle^{\uparrow} \mid c \triangleright (a[y] \triangleright f)) \\ &\mid (\texttt{rcv}\langle (a), (\texttt{ping}), r\rangle^{\uparrow} \mid c \triangleright \texttt{snd}\langle r, \texttt{up}, a\rangle) \\ &\mid (\texttt{rcv}\langle (a), (\texttt{ping}), r\rangle^{\uparrow} \mid f \triangleright \texttt{snd}\langle r, \texttt{down}, a\rangle)\end{aligned}$$

Membrane with Fail-Stop Failures and Recovery. This is a membrane that extends the previous one with the possibility of recovery:

$$\begin{aligned}M_{failr} \triangleq \nu c\, f.M_{trans} \mid c &\mid (\texttt{stop}\langle (a)\rangle^{\uparrow} \mid c \triangleright (a[y] \triangleright f\langle y\rangle)) \\ &\mid (\texttt{rcv}\langle (a), (\texttt{ping}), r\rangle^{\uparrow} \mid c \triangleright \texttt{snd}\langle r, \texttt{up}, a\rangle) \\ &\mid (\texttt{rcv}\langle (a), (\texttt{ping}), r\rangle^{\uparrow} \mid f\langle y\rangle \triangleright \texttt{snd}\langle r, \texttt{down}, a\rangle) \\ &\mid (\texttt{rcv}\langle (a), (\texttt{recover}), r\rangle^{\uparrow} \mid f\langle y\rangle \triangleright a[y] \mid c \mid \texttt{snd}\langle r, \texttt{rcvd}, a\rangle)\end{aligned}$$

3 The Kell Calculus: Syntax and Operational Semantics

3.1 Syntax

The syntax of the Kell calculus, together with the syntax of evaluation contexts, is given below:

$$
\begin{aligned}
P &::= \mathbf{0} \;\mid\; x \;\mid\; \xi \triangleright P \;\mid\; \nu a.P \;\mid\; P \mid P \;\mid\; a[P] \;\mid\; a\langle \widetilde{P} \rangle \\
\xi &::= \bot \;\mid\; J \;\mid\; J \mid a[x] \\
J &::= a\langle \widetilde{u} \rangle^{*} \;\mid\; J \mid J \\
u &::= x \;\mid\; (x) \\
* &::= - \;\mid \uparrow \mid \downarrow \\
\mathbf{E} &::= \cdot \;\mid\; \nu a.\mathbf{E} \;\mid\; a[\mathbf{E}] \;\mid\; P \mid \mathbf{E}
\end{aligned}
$$

Filling the hole $\cdot$ in an evaluation context $\mathbf{E}$ with a Kell calculus term Q results in a Kell calculus term noted $\mathbf{E}\{Q\}$.

We assume an infinite set N of *names*. We let a, b, x, y and their decorated variants range over N. Note that names in the kell calculus act both as name constants and as (name or process) variables. We use $\widetilde{V}$ to denote finite vectors $(V_1, \ldots, V_q)$. Abusing the notation, we equate $\widetilde{V} = (V_1, \ldots, V_n)$ with the word $V_1 \ldots V_n$ and the set $\{V_1, \ldots, V_n\}$. We note $|\widetilde{V}|$ the length n of a vector $\widetilde{V} = (V_1, \ldots, V_n)$.

Terms in the Kell calculus grammar are called *processes*. We note K the set of Kell calculus processes. We let P, Q, R, S, T and their decorated variants range over processes. We call *message* a process of the form $a\langle \widetilde{P} \rangle$. We let M, N and their decorated variants range over messages and parallel composition of messages. We abbreviate a a message of the form $a\langle\rangle$ (i.e. a message with an empty vector of arguments). We call *kell* a process of the form $a[P]$. The name a in a kell $a[P]$ is called the name of the kell. In a kell of the form $a[\ldots \mid a_j[P_j] \mid \ldots \mid Q_k \mid \ldots]$ we call *subkells* the processes $a_j[P_j]$. We call *trigger* a process of the form $\xi \triangleright P$, where ξ is a *receipt pattern* (or *pattern*, for short). A pattern can be a *join pattern* J, or a *control pattern* of the form $J \mid a[x]$, in which the join pattern J may be empty (i.e. $J = \bot$). The empty join pattern, $\bot$, cannot match any message. We note $a\langle \widetilde{u} \rangle$ for $a\langle \widetilde{u} \rangle^{-}$.

In a term $\nu a.P$, the scope extends as far to the right as possible. In a term $\xi \triangleright P$, the scope of $\triangleright$ extends as far to the left and to the right as possible. Thus, $a\langle c \rangle \mid b[y] \triangleright P \mid Q$ stands for $(a\langle c \rangle \mid b[y]) \triangleright (P \mid Q)$. We use standard abbreviations from the the π-calculus: $\nu a_1 \ldots a_q.P$ for $\nu a_1 \ldots . \nu a_q.P$, or $\nu \widetilde{a}.P$ if $\widetilde{a} = (a_1 \ldots a_q)$. By convention, if the name vector $\widetilde{a}$ is empty, then $\nu \widetilde{a}.P \triangleq P$. We also note $\prod_{i \in I} P_i$, $I = \{1, \ldots, n\}$ the parallel composition $(P_1 \mid (\ldots (P_{n-1} \mid P_n) \ldots))$. By convention, if $I = \emptyset$, then $\prod_{i \in I} P_i \triangleq \mathbf{0}$.

A pattern ξ acts as a binder in the calculus. All names x that do not occur within parenthesis () in a pattern ξ are bound by the pattern. We call *pattern variables* (or *variables*, for short) such bound names in a pattern. Variables occurring in a pattern are supposed to be linear, i.e. there is only one occurrence of each variable in a given pattern. Names occurring in a pattern ξ under parenthesis (i.e. occurrences of the form (x) in ξ) are *not* bound in the pattern. We call free pattern names (or free names, for short), names occurring under () in a pattern. The other binder in the calculus is the ν

operator, which corresponds to the restriction operator of the π-calculus. Free names ($\mathtt{fn}$), receiver names ($\mathtt{rn}$), bound pattern variables ($\mathtt{bn}$) and free pattern names ($\mathtt{mn}$) are defined below:

$$
\begin{array}{ll}
\mathtt{fn}(\mathbf{0}) = \emptyset & \mathtt{fn}(x) = \{x\} \\
\mathtt{fn}(\nu x.P) = \mathtt{fn}(P) \setminus \{x\} & \mathtt{fn}(P \mid Q) = \mathtt{fn}(P) \cup \mathtt{fn}(Q) \\
\mathtt{fn}(x[P]) = \mathtt{fn}(P) \cup \{x\} & \mathtt{fn}(J) = \mathtt{rn}(J) \cup \mathtt{mn}(J) \\
\mathtt{fn}(J \triangleright P) = (\mathtt{fn}(P) \setminus \mathtt{bn}(J)) \cup \mathtt{fn}(J) & \\
\mathtt{fn}(x\langle P_1, \ldots, P_n\rangle) = \mathtt{fn}(P_1) \cup \ldots \cup \mathtt{fn}(P_n) \cup \{x\} & \\
\mathtt{fn}(J \mid y[x] \triangleright P) = (\mathtt{fn}(P) \setminus \mathtt{bn}(J) \setminus \{x\}) \cup \{y\} \cup \mathtt{fn}(J) &
\end{array}
$$

$$
\begin{array}{ll}
\mathtt{rn}(a\langle\widetilde{u}\rangle) = a & \mathtt{rn}(J \mid J') = \mathtt{rn}(J) \cup \mathtt{rn}(J') \\
\mathtt{bn}(a\langle\widetilde{u}\rangle) = \mathsf{N} \cap \widetilde{u} & \mathtt{bn}(J \mid J') = \mathtt{bn}(J) \cup \mathtt{bn}(J') \\
\mathtt{mn}(a\langle\widetilde{u}\rangle) = \{x \in \mathsf{N} \mid (x) \in \widetilde{u}\} & \mathtt{mn}(J \mid J') = \mathtt{mn}(J) \cup \mathtt{mn}(J')
\end{array}
$$

We call *substitution* a function $\theta : \mathsf{N} \to \mathsf{N} \uplus \mathsf{K}$ from names to names and Kell calculus processes that is the identity except on a finite set of names. We write $P\theta$ the image under the substitution θ of process P. We note Θ the set of substitutions, and $\mathtt{supp}$ the support of a substitution (i.e. $\mathtt{supp}(\theta) = \{i \in \mathsf{N} \mid \theta(i) \neq i\}$).

Let J be a join pattern, and θ be a substitution such that $\mathtt{bn}(J) \subseteq \mathtt{supp}(\theta)$. We define the image $J\theta$ of J under substitution θ as $\mathtt{cj}(J)\theta$, where $\mathtt{cj}$ is the function defined inductively as:

$$
\begin{array}{lll}
\mathtt{cj}(a) = a & \mathtt{cj}((a)) = a & \mathtt{cj}(\bot) = \mathbf{0} \\
\mathtt{cj}(a\langle\widetilde{w}\rangle) = a\langle\widetilde{\mathtt{cj}(w)}\rangle & & \mathtt{cj}(a\langle\widetilde{w}\rangle^{\downarrow}) = a\langle\widetilde{\mathtt{cj}(w)}\rangle \\
\mathtt{cj}(a\langle\widetilde{w}\rangle^{\uparrow}) = a\langle\widetilde{\mathtt{cj}(w)}\rangle & & \mathtt{cj}(J \mid J') = \mathtt{cj}(J) \mid \mathtt{cj}(J')
\end{array}
$$

We note $P =_\alpha Q$ when two terms P and Q are α-convertible.

Formally, with the syntax presented, the reduction rules in section 3.2 could yield terms of the form $P[Q]$, which are not legal Kell calculus terms (i.e. the syntax does not distinguish between names playing the role of name variables, and names playing the role of process variables). The type system presented in Section 4 rules out such illegal terms.

3.2 Reduction Semantics

The operational semantics of the Kell calculus is defined in the CHAM style [2], via a structural equivalence relation and a reduction relation. The structural equivalence $\equiv$ is the smallest equivalence relation that verifies the rules in Figure 1 and that makes the parallel operator $\mid$ associative and commutative, with $\mathbf{0}$ as a neutral element.

Notice that we do not have structural equivalence rules that deal with scope extrusion beyond a kell boundary (i.e we do not have the Mobile Ambient rule $a[\nu b.P] \equiv \nu b.a[P]$, provided $b \neq a$). As in the Seal calculus, this is to avoid phenomena as illustrated below:

$$(a[x] \triangleright x \mid x) \mid a[\nu b.P] \;\to\; (\nu b.P) \mid (\nu b.P) \qquad (a[x] \triangleright x \mid x) \mid \nu b.a[P] \;\to\; \nu b.P \mid P$$

The reduction relation $\to$ is the smallest binary relation on K^2 that satisfies the rules given in Figure 2, where we assume that $\mathtt{bn}(J) = \mathtt{supp}(\theta)$. Some comments are

$$\nu a.\mathbf{0} \equiv \mathbf{0} \text{ [S.NU.NIL]} \qquad \nu a.\nu b.P \equiv \nu b.\nu a.P \text{ [S.NU.COMM]}$$

$$\frac{a \notin \mathtt{fn}(Q)}{(\nu a.P) \mid Q \equiv \nu a.P \mid Q} \text{ [S.NU.PAR]} \qquad \frac{P =_\alpha Q}{P \equiv Q} \text{ [S.}\alpha\text{]} \qquad \frac{P \equiv Q}{\mathbf{E}\{P\} \equiv \mathbf{E}\{Q\}} \text{ [S.CONTEXT]}$$

Fig. 1. Structural equivalence.

$$\frac{J = J_1 \mid J_2 \qquad J_1 = \prod_{j \in J} a_j \langle \widetilde{w}_j \rangle^\uparrow \neq \bot \qquad \mathtt{fn}(J_1\theta) \cap \widetilde{c} = \emptyset}{J_1\theta \mid b[\nu\widetilde{c}.R \mid J_2\theta \mid (J \triangleright Q)] \to b[\nu\widetilde{c}.R \mid Q\theta \mid (J \triangleright Q)]} \text{ [R.IN]}$$

$$\frac{\begin{array}{c}\widetilde{d} = \widetilde{c} \setminus \widetilde{e} \qquad \widetilde{e} = \widetilde{c} \cap \mathtt{fn}(J_1\theta) \\ \widetilde{e} \cap \mathtt{fn}(J \triangleright Q, J_2\theta, b) = \emptyset \qquad J = J_1 \mid J_2 \qquad J_1 = \prod_{j \in J} a_j \langle \widetilde{w}_j \rangle^\downarrow \neq \bot\end{array}}{J_2\theta \mid (J \triangleright Q) \mid b[\nu\widetilde{c}.R \mid J_1\theta] \to \nu\widetilde{e}.Q\theta \mid (J \triangleright Q) \mid b[\nu\widetilde{d}.R]} \text{ [R.OUT]}$$

$$J\theta \mid a[P] \mid (J \mid a[x] \triangleright Q) \to Q\theta\{P/x\} \mid (J \mid a[x] \triangleright Q) \text{ [R.PASS]}$$

$$\frac{J \neq \bot}{J\theta \mid (J \triangleright Q) \to Q\theta \mid (J \triangleright Q)} \text{ [R.LOCAL]} \qquad \frac{P \to Q}{\mathbf{E}\{P\} \to \mathbf{E}\{Q\}} \text{ [R.CONTEXT]}$$

$$\frac{P' \equiv P \qquad P \to Q \qquad Q \equiv Q'}{P' \to Q'} \text{ [R.STRUCT]}$$

Fig. 2. Reduction Relation.

in order. Rules R.IN an R.OUT take into account the presence of restrictions inside kells, since restricted names cannot be automatically extruded out of kells through the structural equivalence. Rule R.OUT explicitly extrudes restricted names that are communicated outside a kell boundary. Note that names that are not communicated are not extruded. Rules R.IN and R.OUT govern the crossing of kell boundaries. Note that only messages may cross a kell boundary. In rule R.IN, a trigger receives messages from the local environment as well as from the outside of the enclosing kell. In rule R.OUT, a trigger receives messages from the local environment as well as from a subkell. Rule R.PASS allows the passivation of a subkell, possibly upon receipt of messages from the local environment. In rules R.IN and R.OUT, note that the join pattern J_2 may be empty. Likewise, in rule R.PASS, the join pattern J may be empty.

4 Type System

As pointed out in the introduction, the unicity of kell names is an important property to enforce in order to ensure an efficient implementation of the calculus. For instance, a kell a modelling a computing site interconnected via a wide-area network such as the Internet would have triggers of the form $\mathtt{rcv}\langle(a), (b), \widetilde{x}\rangle \mid \ldots \triangleright P$ with a corresponding e.g. to a wide-area network address. In this setting, the name a must be unique, at least within the context of the enclosing environment (which models the behavior of the network). Enforcing the unicity of kell names, however, is difficult in presence of higher-order communication and kell passivation. For instance, assume that a trigger $\mathtt{twice}\langle x\rangle \triangleright x \mid x$ is defined. Then a trigger of the form $a[x] \triangleright \mathtt{twice}\langle a[x]\rangle$ would lead to the illicit duplication of kell a.

We present in this section a type system for the kell calculus that enforces the unicity of kell names. More precisely, the type system enforces the unicity of *active* kells. A kell $a[Q]$ is said to be active in P (and P is said to contain the active kell a) if $P = \mathbf{E}\{a[Q]\}$ and $a[Q]$ is not under a scope restriction for a. The general idea, borrowed from the M-calculus type system, is to define the type of a process P as a multiset Δ that represents an upper bound on the multiset of names of kells that may be or may become active in P. Intuitively, a process will therefore be well-typed if its type Δ ends up being a set.

The syntax of types is given below:

$$\begin{aligned}
\sigma &::= \mathtt{kell}(w)_{\Delta \to \Delta'} \mid \langle \widetilde{\sigma}\rangle_{\Delta} \mid \langle \widetilde{\sigma}\rangle^{+}_{\Delta} \mid \Delta \\
\Delta &::= \emptyset \mid \rho \mid \delta \mid a \mid \Delta, \Delta \\
w &::= a \mid \delta \mid \emptyset \\
s &::= \forall \widetilde{\rho}\widetilde{\delta}.\sigma
\end{aligned}$$

A type σ can be a *process type* Δ, a *kell name type* $\mathtt{kell}(w)_{\Delta \to \Delta'}$, a *channel type* $\langle \widetilde{\sigma}\rangle_{\Delta}$, or a *sendable channel type* $\langle \widetilde{\sigma}\rangle^{+}_{\Delta}$. A channel of type $\langle \widetilde{\sigma}\rangle_{\Delta}$ can receive messages with arguments of types $\widetilde{\sigma}$, and the receipt of messages on this channel leads to the creation of kells with names in Δ. A sendable channel type types a receiver variable that can be instantiated to a just received name. We note $\langle \widetilde{\sigma}\rangle^{(+)}_{\Delta}$ to denote a type that can be a channel type $\langle \widetilde{\sigma}\rangle_{\Delta}$ or a sendable channel type $\langle \widetilde{\sigma}\rangle^{+}_{\Delta}$.

An active kell name a, which hosts subkells whose names are in Δ, and whose passivation leads to the creation of kells whose names are in Δ', has type $\mathtt{kell}(a)_{\Delta \to \Delta'}$. This is because a kell name can be used both as the name of an active kell and as the name of a special channel used to passivate the kell of the same name (via rule R.PASS). Since kell names are also variables, one must allow name type variables δ as argument of kell name types. No kell name may have type $\mathtt{kell}(\emptyset)_{\Delta \to \Delta'}$ (these types are introduced for technical reasons).

We use $\forall \widetilde{\rho}\widetilde{\delta}.\sigma$ to denote a type scheme in which name type variables $\widetilde{\delta}$ and multiset variables $\widetilde{\rho}$ are generalized. To define the type system, we consider an extended syntax for the calculus where new names are annotated with their type scheme. Thus we write $\nu a : s.P$ instead of $\nu a.P$, where s is a type scheme. The notion of free names is modified to take into account the new syntax: $\mathtt{fn}(\nu y : s.P) = \mathtt{fn}(P, s) \setminus \{y\}$. The structural congruence rules S.NU.NIL, S.NU.COMM and S.NU.PAR are modified thus:

S.NU.NIL	$\nu a : s.\mathbf{0} \equiv \mathbf{0}$	
S.NU.COMM	$\nu a : s.\nu b : s'.P \equiv \nu b : s'.\nu a : s.P$	if $a \notin \mathtt{fn}(s')$ and $b \notin \mathtt{fn}(s)$
S.NU.PAR	$\nu a : s.P \mid Q \equiv (\nu a : s.P) \mid Q$	if $a \notin \mathtt{fn}(Q)$

Multisets Δ can include names a, name type variables δ, and multiset variables ρ. We use several operations on multisets. Relation $\subseteq$ is the standard multiset inclusion. Δ, Δ' is the union of multisets Δ and Δ'. Multiset $\Delta \setminus a$ is the multiset Δ minus a single occurrence of name a. $\Delta \sqcup \Delta'$ is the smallest multiset (in terms of inclusion) that contains both Δ and Δ'.

We define the subtyping relation $\leq$ (where $\widetilde{\sigma}$ and $\widetilde{\sigma'}$ are vectors of the same length n), which is the smallest reflexive relation obeying the rules below:

$$\begin{aligned}
\Delta \leq \Delta' &\Leftarrow \Delta \subseteq \Delta' \\
\langle\widetilde{\sigma}\rangle_\Delta \leq \langle\widetilde{\sigma'}\rangle_{\Delta'} &\Leftarrow \forall i \in \{1,\ldots,n\}, \sigma'_i \leq \sigma_i \ \wedge \ \Delta \subseteq \Delta' \\
\langle\widetilde{\sigma}\rangle^+_\Delta \leq \langle\widetilde{\sigma'}\rangle_{\Delta'} &\Leftarrow \forall i \in \{1,\ldots,n\}, \sigma'_i \leq \sigma_i \ \wedge \ \Delta \subseteq \Delta' \\
\mathtt{kell}(w_1)_{\Delta_1\to\Delta_2} \leq \mathtt{kell}(w_2)_{\Delta'_1\to\Delta'_2} &\Leftarrow w_1 = w_2 \ \wedge \ \Delta'_1 \subseteq \Delta_1 \ \wedge \ \Delta_2 \subseteq \Delta'_2
\end{aligned}$$

The intuition behind the subtyping relation is that it is safe (with respect to the unicity of kell names) to replace a process with a process that contains fewer active kells.

We use Γ and its decorated variants to denote type environments, i.e. finite mappings between names and type schemes. We define the set of free names and of free type variables below:

$$\begin{array}{ll}
\mathtt{fn}(\emptyset) = \emptyset & \mathtt{fv}(\emptyset) = \emptyset \\
\mathtt{fn}(\rho) = \emptyset & \mathtt{fv}(\rho) = \{\rho\} \\
\mathtt{fn}(\delta) = \emptyset & \mathtt{fv}(\delta) = \{\delta\} \\
\mathtt{fn}(a) = a & \mathtt{fv}(a) = \emptyset \\
\mathtt{fn}(\Delta,\Delta') = \mathtt{fn}(\Delta) \cup \mathtt{fn}(\Delta') & \mathtt{fv}(\Delta,\Delta') = \mathtt{fv}(\Delta) \cup \mathtt{fv}(\Delta') \\
\mathtt{fn}(\mathtt{kell}(w)_{\Delta\to\Delta'}) = \mathtt{fn}(w) \cup \mathtt{fn}(\Delta,\Delta') & \mathtt{fv}(\mathtt{kell}(w)_{\Delta\to\Delta'}) = \mathtt{fv}(w) \cup \mathtt{fv}(\Delta,\Delta') \\
\mathtt{fn}(\langle\widetilde{\sigma}\rangle^{(+)}_\Delta) = \mathtt{fn}(\sigma_1) \cup \ldots \cup \mathtt{fn}(\sigma_n) \cup \mathtt{fn}(\Delta) & \mathtt{fv}(\langle\widetilde{\sigma}\rangle^{(+)}_\Delta) = \mathtt{fv}(\sigma_1) \cup \ldots \cup \mathtt{fv}(\sigma_n) \cup \mathtt{fv}(\Delta) \\
\mathtt{fn}(\forall\widetilde{\beta}.\sigma) = \mathtt{fn}(\sigma) & \mathtt{fv}(\forall\widetilde{\beta}\sigma) = \mathtt{fv}(\sigma) \setminus \widetilde{\beta} \\
\mathtt{fn}(\Gamma) = \cup_{x\in dom(\Gamma)}\mathtt{fn}(\Gamma(x)) & \mathtt{fv}(\Gamma) = \cup_{x\in dom(\Gamma)}\mathtt{fv}(\Gamma(x))
\end{array}$$

Type judgments take the following form: $\Gamma \vdash P : \sigma$, where Γ is an environment, P is a process, and σ is a type. The type system is defined by the rules in Figure 3. They make use of the `Inst` operator, that takes a type scheme and returns a type where the generalized name type variables and multiset variables have been instantiated to names and multisets, respectively. A type environment Γ is said to a be *good* if $\mathtt{fn}(\Gamma) = \{x \in dom(\Gamma) \mid \Gamma(x) = \forall\widetilde{\beta}.\mathtt{kell}(x)_{\Delta\to\Delta'}\}$.

The typing rules use auxiliary functions which we now define. To deal with sendable receivers, we introduce a partition of the set of names, N: we define a set V such that $\mathsf{V} \subseteq \mathsf{N}$. If $a \in \mathsf{V}$, then a must be of type sendable. This is formalized in the definition of predicate `Pred` below. Assume that $(a_i, \forall\widetilde{\beta_i}.\langle\sigma_i^{1..m_i}\rangle_{\Delta_i}) \in \Gamma$ and $J = a_1\langle u_{1j}^{1..m_1}\rangle \mid \ldots \mid a_n\langle u_{nj}^{1..m_n}\rangle$. We define the function `Extract` by:

$$\mathtt{Extract}(\Gamma, J) = \{x_{ij} : \sigma_{ij} \mid x_{ij} \in \mathrm{bn}(J)\}$$

We define the predicate `Pred` by:

$$\begin{aligned}
\mathtt{Pred}(\Gamma, J) = {} & \forall i,i' \in \{1..n\}.\forall j \in \{1..m_i\}.\forall j' \in \{1..m_{i'}\} \\
& (x_{ij}, x_{i'j'} \in \mathrm{mn}(J) \wedge (x_{ij} = x_{i'j'})) \implies \sigma_{ij} = \sigma_{i'j'} \\
& \wedge\ \forall i \in \{1..n\}.\forall j \in \{1..m_i\}.x_{ij} \in \mathrm{mn}(J) \implies (\sigma_{ij} \neq \Delta) \\
& \wedge\ \forall i \in \{1..n\}.fv(\Gamma) \cap \widetilde{\beta_i} = \emptyset \\
& \wedge\ \forall i,j \in \{1..n\}.i \neq j \implies \widetilde{\beta_i} \cap \widetilde{\beta_j} = \emptyset \\
& \wedge\ \forall x \in \mathrm{bn}(J), x \in \mathsf{V} \\
& \wedge\ a_i \in \mathsf{V} \implies (a_i, \langle\sigma_i^{1..m_i}\rangle^+_{\Delta_i}) \in \Gamma
\end{aligned}$$

$$\frac{\Gamma \text{ good} \qquad (x, s = \forall\widetilde{\beta}.\sigma) \in \Gamma, \qquad \sigma\theta = \mathtt{Inst}(s) \qquad \mathtt{fn}(ran(\theta)) \subseteq \mathtt{fn}(\Gamma)}{\Gamma \vdash x : \sigma\theta} \text{[T.VAR]}$$

$$\frac{\Gamma \text{ good}}{\Gamma \vdash \mathbf{0} : \emptyset} \text{[T.NIL]} \qquad \frac{\Gamma \vdash P_1 : \Delta_1 \qquad \Gamma \vdash P_2 : \Delta_2}{\Gamma \vdash P_1 \mid P_2 : \Delta_1, \Delta_2} \text{[T.PAR]}$$

$$\frac{\begin{array}{c} s = \forall\widetilde{\beta}.\langle\widetilde{\sigma}\rangle_\Delta \vee s = \langle\widetilde{\sigma}\rangle^+_\Delta \\ fv(s) = \emptyset \qquad \Gamma,\ r : s \vdash P : \Delta' \qquad \mathtt{fn}(s) \subseteq \mathtt{fn}(\Gamma) \end{array}}{\Gamma \vdash \nu r : s.P : \Delta'} \text{[T.CHAN.KELL]}$$

$$\frac{\begin{array}{c} s = \forall\widetilde{\beta}.\mathtt{kell}(a)_{\rho\to\Delta'} \qquad fv(s) = \emptyset \qquad \rho \notin \Delta' - \rho \\ \Gamma,\ a : s \vdash P : \Delta, \qquad a \notin \Delta - a \qquad \mathtt{fn}(\Delta') \subseteq \mathtt{fn}(\Gamma) \cup \{a\} \qquad a \notin \mathtt{fn}(\Gamma) \end{array}}{\Gamma \vdash \nu a : s.P : \Delta - a} \text{[T.CHAN]}$$

$$\frac{\Gamma \vdash P : \Delta_0 \qquad \Gamma \vdash a : \mathtt{kell}(w)_{\Delta\to\Delta'} \qquad \Delta_0 \leq \Delta}{\Gamma \vdash a[P] : (w,\ \Delta_0) \sqcup \Delta'} \text{[T.KELL]}$$

$$\frac{\Gamma \vdash a : \langle\widetilde{\sigma}\rangle^{(+)}_\Delta \qquad \Gamma \vdash P_i : \sigma'_i \qquad \sigma'_i \leq \sigma_i}{\Gamma \vdash a\langle\widetilde{P}\rangle : \Delta} \text{[T.MSG]}$$

$$\frac{\begin{array}{c} J = a_1\langle u_{1j}^{1..m_1}\rangle^* \mid \ldots \mid a_n\langle u_{nj}^{1..m_n}\rangle^* \\ \Gamma' = \mathtt{Extract}(\Gamma, J) \qquad \mathtt{Pred}(\Gamma, J) \\ (a_i, \forall\widetilde{\beta_i}.\langle\sigma_i^{1..m_i}\rangle^{(+)}_{\Delta_i}) \in \Gamma \qquad \Gamma,\ \Gamma' \vdash P : \Delta \qquad \Delta \leq \Delta_1, \ldots, \Delta_n \\ \Gamma \vdash a_i : \langle\tau_i^{1..m_i}\rangle^{(+)}_{\Delta'_i} \qquad \forall x_{ij} \in \mathtt{mn}(J).\Gamma \vdash x_{ij} : \tau'_{ij} \qquad \tau'_{ij} \leq \tau_{ij} \end{array}}{\Gamma \vdash J \triangleright P : \emptyset} \text{[T.TRIG.MSG]}$$

$$\frac{\begin{array}{c} J = a_1\langle u_{1j}^{1..m_1}\rangle^* \mid \ldots \mid a_n\langle u_{nj}^{1..m_n}\rangle^* \\ \Gamma' = \mathtt{Extract}(\Gamma, J) \qquad \mathtt{Pred}'(\Gamma, J, a) \qquad (a, \forall\widetilde{\beta}.\mathtt{kell}(w)_{\Delta\to\Delta_0}) \in \Gamma \\ (a_i, \forall\widetilde{\beta_i}.\langle\sigma_i^{1..m_i}\rangle^{(+)}_{\Delta_i}) \in \Gamma \qquad \Gamma,\ \Gamma', x : \Delta \vdash P : \Delta' \qquad \Delta' \leq \Delta_0, \ldots, \Delta_n \\ \Gamma \vdash a_i : \langle\tau_i^{1..m_i}\rangle^{(+)}_{\Delta'_i} \qquad \forall x_{ij} \in \mathtt{mn}(J).\Gamma \vdash x_{ij} : \tau'_{ij} \qquad \tau'_{ij} \leq \tau_{ij} \end{array}}{\Gamma \vdash J \mid a[x] \triangleright P : \emptyset} \text{[T.TRIG.PASS]}$$

Fig. 3. Typing rules.

Assume now that $(a, \forall\widetilde{\beta}.\mathtt{kell}(w)_{\Delta\to\Delta_0}) \in \Gamma$. We define the predicate $\mathtt{Pred}'$ by:

$$\mathtt{Pred}'(\Gamma, J, a) = \mathtt{Pred}(\Gamma, J)\ \wedge\ (fv(\Gamma) \cap \widetilde{\beta} = \emptyset)\ \wedge\ (a \notin \mathsf{V})$$

Some comments on these typing rules are in order. In rule T.MSG, the type of channel a is in fact a type scheme. The condition $\Gamma \vdash a : \langle\widetilde{\sigma}\rangle_\Delta$ provides an instance of this type scheme. Rule T.MSG requires arguments P_i of a message on channel a to have types which are subtypes of the expected argument types. Because of rules T.TRIG.MSG and T.TRIG.PASS, a trigger always has type $\emptyset$ (a trigger does not exhibit any active kell). The premises in both rules deal with the two sorts of names in a join pattern (variables and free names). In the case of a free name a (which occurs as (a) in the pattern), one must ensure that it is identically typed in all of its occurrences (first clause in the definition of $\mathtt{Pred}$), and that its type is not a process type (condition $\sigma_{ij} \neq \Delta$ in the second clause of the definition of $\mathtt{Pred}$). Note that the part of the premises that deals with free names is actually similar to the premises in rule T.MSG, since T.MSG only deals with free names and processes. Typing rules T.TRIG.MSG and T.TRIG.PASS may seem complex but they are in fact very close to the typing rule for Join calculus definitions: the guarded process is typed in an environment extended with formal parameters, and the result is checked to create fewer kells than advertised by the channel types. Every defined channel name that is a variable is checked to have a sendable channel type in the environment. The additional hypotheses check that the type schemes associated with channels (and the passivated kell name in T.TRIG.PASS) are consistent with the typing environment: no generalized variable may occur free in the environment, nor be shared by two channels (or a channel and the passivated kell name in T.TRIG.PASS).

The soundness of the type system is characterized by the following definitions and theorems, where a good type environment Γ is said to be *well-formed* if all pairs in Γ are of one of the following forms: $x : \langle\widetilde{\sigma}\rangle^+_\Delta$, $x : \forall\widetilde{\beta}.\langle\widetilde{\sigma}\rangle_\Delta$, or $x : \forall\widetilde{\beta}.\mathtt{kell}(x)_{\rho\to\Delta}$ with $\rho \notin \Delta - \rho$. A process P is said to have *failed* if $P = \mathbf{E}\{Q\}$, with Q containing two active localities bearing the same name. A process P is said to be *faulty* if $P \to^* Q$ with Q failed.

Theorem 1. *If $\Gamma \vdash P : \sigma$ with Γ well-formed, and $P \equiv Q$, then $\Gamma \vdash Q : \sigma$.*

Theorem 2 (Subject Reduction). *If $\Gamma \vdash P : \sigma$ with Γ well-formed, and $P \to Q$, then there exists σ' such that $\sigma' \leq \sigma$ and $\Gamma \vdash Q : \sigma'$.*

Theorem 3 (Progress). *If $\Gamma \vdash P : \Delta$ with Γ well-formed, and Δ is a set containing only kell names, then the process P is not faulty.*

We now discuss some features and limitations of the type system. Note first that, because of the constraint $a \notin \mathsf{V}$ in the definition of predicate $\mathtt{Pred}'$, an expression such as $a\langle y\rangle \triangleright (y[x] \triangleright P)$ is not typable. In other terms, one cannot instantiate a kell name with a received name. Like the type system of the M-calculus defined in [14] from which it is inspired, our type system simulates dependent types using polymorphism and name type variables (type variables that represent kell names), since kell names may occur in types. Consider now some simple examples. The process $(a\langle y\rangle \triangleright b[y]) \mid a\langle\mathbf{0}\rangle \mid a\langle\mathbf{0}\rangle$ is faulty, since in the environment $\Gamma = a : \forall\rho.\langle\rho\rangle_{b,\rho},\ b : \forall\rho'.\mathit{dom}(b)_{\rho'\to\emptyset}$, we get the type judgment $\Gamma \vdash (a\langle y\rangle \triangleright b[y]) \mid a\langle\mathbf{0}\rangle \mid a\langle\mathbf{0}\rangle : b, b$. This is an example of a process which is correctly flagged as faulty by our type system. As another example, the process $a\langle y\rangle \mid b\langle z\rangle \triangleright y \mid z$ is correct (non-faulty) and is indeed typable: with the environment $\Gamma = a : \forall\rho.\langle\rho\rangle_\rho,\ b : \forall\rho'.\langle\rho'\rangle_{\rho'}$, we get $\Gamma \vdash a\langle y\rangle \mid b\langle z\rangle \triangleright y \mid z : \emptyset$. On the other hand, consider the process $T = a\langle y\rangle \triangleright (\nu t : \langle\rangle_\emptyset.(t \mid b\langle z\rangle \triangleright y \mid z) \mid t)$. This process is correct since it does not duplicate kells it receives (it just instantiates a process received on a once). However this process is not typable with our type system. Indeed, the type system is too coarse in that respect since it deals with process T in the same way than with process $S = a\langle y\rangle \triangleright (b\langle z\rangle \triangleright y \mid z)$, which is indeed faulty since it may lead to an indefinite replication of active kell names received in the y argument. It is not clear how this limitation can be lifted.

5 Conclusion

We have introduced the Kell calculus, a new process calculus with hierarchical localities, strictly local actions, higher-order communication and locality passivation. Like the M-calculus, the Kell calculus allows an encoding of different forms of locality membranes, including localities with different forms of failures. The Kell calculus, however, appears simpler than the M-calculus, and does not rely on complex routing rules in contrast to the M-calculus.

The Kell calculus shares the local character of its actions with the Seal calculus [6]. Indeed, as in the Seal calculus, primitive actions in our calculus include local communications and communications across a single locality boundary. In contrast to Seal, however, our communications are higher-order, whereas Seal distinguishes between first-order communications on the one hand and migrating and replicating localities on the other hand. The choice in Seal to eschew higher-order communication was made primarily with a view to simplify its underlying theory. However, as the results in [6] reveal, the higher-order character of the migrate and replicate primitives in Seal already poses some problems (e.g. with respect to a complete characterization of contextual equivalence). With the Kell calculus higher-order pirmitives, we gain the ability to handle directly passivated process states. This allows for instance a direct modelling of such failure behaviors as fail-stop with recovery, a behaviour which would be less straightforward to model in Seal (seals can be replicated and destroyed but they cannot be passivated and reactivated; it is possible to place Seals in opaque membranes to simulate passivation but this is not entirely satisfactory since one can allow observation of passivated states – e.g. in the form of checkpoints). Another perceived advantage of the higher-order character of the Kell calculus over Seal is the potential to extend the calculus with multi-stage programming along e.g. the lines of MetaKlaim [8].

The type system we introduced for the Kell calculus is directly inspired by the M-calculus type system [14]. Because the calculus is simpler, with less constructs, the resulting type system is also simpler. Whether the two type systems are comparable (notably with respect to the amount of correct processes they fail to type) is unclear, however. In particular, it is not clear whether a typed encoding of the M-calculus in the Kell calculus would yield a similar type system for the M-calculus as the original one.

To the best of our knowledge, the dual use which is made in the Kell calculus of the locality construct $a[P]$, both as a locus for computation and as a handle for controlling the execution of a located process, is new. The examples provided in this paper, together with the encodings of Mobile Ambients and of the Distributed Join calculus given in [15], show that a single (higher-order) objective passivation construct is sufficient to capture the variety of subjective migration primitives which have been proposed recently, in ambient calculi and other distributed process calculi. At the same time, this construct is powerful enough to model different forms of failures, including fail-stop failures with recovery, an important requirement for practical distributed programming.

Much work remains to be done, however, to assess the foundational character of the calculus with respect to distributed programming. The following issues seem worth considering:

- Developing a bisimulation theory for the Kell calculus. A characterization of contextual equivalence by means of a higher-order bisimulation seems highly nontrivial because of the passivation construct.
- Developing type systems for the Kell calculus. Numerous type systems have been developed for mobile Ambients and their variants. It would be interesting to transfer these results to the Kell calculus (in particular the ones dealing with resource and security constraints).
- Introducing the possibility to share processes among different kells. If one considers a kell (or locality) not only as a locus of computation but also as a component, sharing among kells appears as an important practical requirement. However, sharing raises considerable difficulties, which are very much related to the aliasing problem in object-oriented programming.

References

1. R. Amadio. An asynchronous model of locality, failure, and process mobility. Technical report, INRIA Research Report RR-3109, INRIA Sophia-Antipolis, France, 1997.
2. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science, vol. 96*, 1992.
3. M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *4th International Symposium on Theoretical Aspects of Computer Software (TACS)*, 2001.
4. M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication Interference in Mobile Boxed Ambients. In *Proceedings of the 22nd Conference on Foundations of Software Technology and Theoretical Computer Science (FST-TCS '02*, volume LNCS 2556. Springer, 2002.
5. L. Cardelli and A. Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures, M. Nivat (Ed.), Lecture Notes in Computer Science, Vol. 1378*. Springer Verlag, 1998.
6. G. Castagna and F. Zappa. The Seal Calculus Revisited. In *In Proceedings 22th Conference on the Foundations of Software Technology and Theoretical Computer Science*, number 2556 in LNCS. Springer, 2002.
7. M. Coppo, M. Dezani-Ciancaglini, E. Giovannetti, and I. Salvo. $\mathbf{M}^3$: Mobility types for mobile processes in mobile ambients. In *CATS 2003)*, volume 78 of *ENTCS*, 2003.
8. G. Ferrari, E. Moggi, and R. Pugliese. MetaKlaim: A Type-Safe Multi-Stage Language for Global Computing. *to appear in Mathematical Structures in Computer Science*, 2003.
9. C. Fournet, J.J. Levy, and A. Schmitt. An asynchronous distributed implementation of mobile ambients. In *Proceedings of the International IFIP Conference TCS 2000, Sendai, Japan, Lecture Notes in Computer Science 1872*. Springer, 2000.
10. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. Technical report, Technical Report 2/98 – School of Cognitive and Computer Sciences, University of Sussex, UK, 1998.
11. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000)*, 2000.
12. M. Merro and M. Hennessy. Bisimulation congruences in safe ambients. In *29th ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon, 16-18 January*, 2002.

13. D. Sangiorgi and A. Valente. A Distributed Abstract Machine for Safe Ambients. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming*, volume 2076 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2001.
14. A. Schmitt and J.B. Stefani. The M-calculus: A Higher-Order Distributed Process Calculus. In *Proceedings 30th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2003.
15. J.B. Stefani. A Calculus of Kells. In *Proceedings 2nd International Workshop on Foundations of Global Computing*, 2003.
16. D. Teller, P. Zimmer, and D. Hirschkoff. Using Ambients to Control Resources. In *Proceedings CONCUR 02*, 2002.
17. J. Vitek and G. Castagna. Towards a calculus of secure mobile computations. In *Proceedings Workshop on Internet Programming Languages, Chicago, Illinois, USA, Lecture Notes in Computer Science 1686, Springer*, 1998.
18. P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure. *IEEE Concurrency, vol. 8, no 2*, 2000.
19. N. Yoshida and M. Hennessy. Assigning types to processes. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2000.

A Calculus for Long-Running Transactions

Laura Bocchi, Cosimo Laneve, and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura A.Zamboni 7, I-40127 Bologna, Italy
{bocchi,laneve,zavattar}@cs.unibo.it

Abstract. We study *long-running transactions* in open component-based distributed applications, such as Web Services platforms. Long-running transactions describe time-extensive activities that involve several distributed components. Henceforth, in case of failure, it is usually not possible to restore the initial state, and firing a compensation process is preferable. Despite the interest of such transactional mechanisms, a formal modeling of them is still lacking. In this paper we address this issue by designing an extension of the asynchronous π-calculus with long-running transactions (and sequences) – the $\pi\mathsf{t}$-*calculus*. We study the practice of $\pi\mathsf{t}$-calculus, by discussing few paradigmatic examples, and its theory, by defining a semantics and providing a correct encoding of $\pi\mathsf{t}$-calculus into asynchronous π-calculus.

1 Introduction

Web Services technology intend to provide standard mechanisms for describing the interface and the services available on the web, as well as protocols for locating such services and invoking them (*cf.* WSDL standard [7]). A relevant feature, which is usually overlooked by Web Services, is the mechanism for their reuse when complex tasks are carried out. It is often the case, in business-to-business processes, to define new processes out of finer-grained subtasks that are likely available as Web Services. Therefore it is reasonable to forecast an extension of the Web Service standard, which supports the definition of complex services out of simpler ones – the so called *Web Services choreography.* Indeed, several proposals that describe Web Services choreography have been already set up: BPML [6] by BPMI.org, XLANG [18] and BizTalk [14] (a visual specification environment for XLANG) by Microsoft, WSFL [13] by IBM, BPEL4WS [9] by a consortium grouping BEA, IBM, Microsoft, and others), etc. The W3C Web Services Choreography Working Group is expected to present the Recommendation for the Web Services choreography specification for November 2003.

Most of these proposals use long-running transactions as a mechanism for describing *loosely-coupled* activities. On the contrary, traditional transactions in databases have been devised to compose *tightly-coupled* activities. These transactions are addressed by the keyword "ACID" to refer to the four properties that they guarantee – Atomicity, Consistency, Isolation, and Durability.

When the activities involved in a transaction are loosely-coupled, the ACID properties adapt badly. In particular, serializability (Isolation) requires that different activities have the same effect whether they are executed in sequence or in parallel. Usually, this is enforced by locking the resources used by each activity until the transaction commits. In the context of Web Services, the processes involved in a transaction may belong to different companies, and there is no chance to lock resources of other companies. Additionally, commercial transactions usually last long periods of time, even months, and it is not feasible and not reasonable to block resources so long. For similar reasons, Atomicity ("all or nothing"), and the two-phase commit protocol to enforce it, become impracticable in the case of long-term commercial transactions.

One immediately ends up in weakening the notion of rollback: in a business process, the provider might decide that rollback will not cancel all the operations carried out. The cancellation of an airplane booking, for instance, may lead to the payment of a fee; the interactions with non-transactional resources, which do not support an "absolute" mechanism of rollback, make failures extremely complicated, and to be dealt with ad-hoc ways. Overall, web transactions or, better, long-running transactions, miss serializability, due to the absence of locks on resources, and possess a lightweight notion of atomicity enforced by ad-hoc rollbacks, called *compensations.*

Despite the interest in long-running transactions, there is not yet a common agreement about their meaning. The above proposals for Web Services choreography have slightly different interpretations of long-running transactions, which may be hardly pointed out due to the informal nature of the documents, and to the complexity of parsing implementations (see the following subsection of related works).

In this paper we propose a formal approach to the description of long-running transactions. In particular, we consider Microsoft BizTalk, and the long-running transactions therein, and we define a formal model and an implementation. The other notions of long-running transactions could be defined as well in similar ways, and compared with the semantics of BizTalk. This is a considerable future work.

Long-running transaction in BizTalk have two associated activities: the *failure* process and the *compensation* process. There are two kinds of transactions: those without inner transactions and the others. The first case is simpler: if the transaction fails, the failure process is executed. In the second case, if a transaction with inner transactions fails, the compensations of the inner transactions must be executed before activating the failure process of the enclosing transaction. Namely, after failure, the compensations can be activated in any possible order, independently of the order in which the corresponding transactions completed. Therefore, the programmer must explicitly describe inter-dependencies among compensations, to avoid undesired schedules by the run-time system.

In our formal analysis we proceed as follows. We introduce a core language with BizTalk transactions, the $\pi\mathsf{t}$-calculus, and we define its operational semantics. It turns out that $\pi\mathsf{t}$-calculus is an extension of the asynchronous variant [4] of the π-calculus [15]. We then report some paradigmatic examples of long-term business activities. Afterwards we implement $\pi\mathsf{t}$-calculus in the asynchronous π-calculus, thus providing a further definition of the meaning of BizTalk transaction. Finally we demonstrate that this meaning conforms with the operational semantics of πt-calculus.

The choice to extend π-calculus with transactions, rather than other process calculi, is due to XLANG, which is the language implementing the orchestration services of BizTalk and whose definition has been strongly influenced by π-calculus. As regards our compilation, it is compositional, and therefore easily amenable to a distributed implementation. In this sense, our result may be read as a correctness proof of a distributed implementation of long-running transactions.

1.1 Related Works

Long-running transactions have been introduced in "data processing applications" in [12, 11], where they were called *saga*. Sagas are possibly nested processes with a monitor and a compensation. When a saga fails, if it doesn't contain nested sagas, the compensation of the previously completed sagas are executed in the reverse order of composition. If the saga contains nested sagas, and a nested saga fails, the compensations of the successfully terminated sibling sagas are executed in the reverse order of commit. In this case, the monitor of the enclosing saga is notified of the abort.

Web services languages, such as WSFL [13], XLANG [18], and BPEL [9], support long-running transactions with flexible failure managements. In these languages, an aborted transaction raises an exception that is catched by a suitable programmable handler. When the handler is omitted, a failure activates the compensation of the completed transactions in the reverse order of commit. This mismatches with BizTalk semantics.

Other contributions to the definition of long-running transactions arise in "web transaction protocols", which define models for orchestrating loosely-coupled web services by means of a defined set of transaction messages. In particular, the W3C Tentative Hold Protocol (THP) [16], uses an architecture with two actors: clients and resources. The clients send temptative holds to resources, requesting information about existing holds, cancelling holds, and retrieving stored informations. The resources use a programmable "rule engine entity" for their management. Another protocol is the OASIS Business Transaction Protocol (BTP) [10], which supports nesting of transactions as in BizTalk. OASIS introduces the notion of cohesion to bear different outcomes from participants to a transaction. In partiular, a transaction may succeed even if some of its inner transactions fail, provided that a minimal number of sub-transactions have succeeded.

1.2 Structure of the Paper

In Section 2 we define the syntax and operational semantics of the πt-calculus. In Section 3 we report examples of long-running transactions described in πt-calculus. In Section 4 we discuss the encoding of πt-calculus into asynchronous

π-calculus. Section 5 contains some conclusive remarks and a comparison with the related literature.

2 Syntax and Semantics of πt-Calculus

2.1 The Syntax

The syntax of the πt-calculus uses a countable set of names $\mathcal{N}$ ranged over by x, y, u, v, ... Tuples of names are written $u_i^{i\in 1..p}$ or simply $\widetilde{u}$. In order to support process constant definitions, we assume given a set of process constants ranged over by K, K', ...

A process (or an agent) in πt-calculus is defined by the following syntax:

$$
\begin{array}{lll}
P ::= & \mathsf{done} & \text{success} \\
 & \mid \mathsf{abort} & \text{error} \\
 & \mid \overline{x}\langle\widetilde{u}\rangle & \text{output} \\
 & \mid x(\widetilde{u}).P & \text{input} \\
 & \mid P \mid P & \text{parallel} \\
 & \mid P;P & \text{sequence} \\
 & \mid (x)P & \text{new} \\
 & \mid K(\widetilde{u}) & \text{invocation} \\
 & \mid \mathsf{t}(P,\, P,\, P,\, P) & \text{transaction}
\end{array}
$$

The new operator $(x)P$ and the input prefix operator $x(\widetilde{u}).P$ are binders for the names x and $\widetilde{u}$, respectively. We omit the standard definitions of *free variables* and *bound variables* of processes, noted $\mathsf{fv}(\cdot)$ and $\mathsf{bv}(\cdot)$, respectively. For each process constant K we assume given a single constant definition $K(\widetilde{u}) \overset{def}{=} P$ where $\widetilde{u}$ is a sequence of pairwise different names and P is a process. If the list of parameters is empty, we omit the surrounding parenthesis, e.g. we use K instead of $K()$.

The processes output, parallel, new, and invocation, are as usual [15]. To enlight the notation we sometime write $(x_1 \dots x_n)P$ instead of $(x_1)\dots(x_n)P$. As usual, we also abbreviate the parallel of P_i for $i \in I$, where I is a finite set, with $\prod_{i\in I} P_i$. The processes done and abort do nothing except manifesting a successful or erroneous termination – to be used inside a transactional context. The input process has an explicit continuation. The sequence process $P; Q$ forces a temporal order between the two operands: process Q will be activated only after successful completion of P. It is worth to notice that our sequential composition operator, even if similar to other operators of the tradition of process algebras (see e.g. the operator $\cdot$ of ACP [2]), is new due to the presence of two different kinds of process termination, represented by the processes done and abort respectively. As we will discuss in the following, these two processes are treated differently when they appear as first operand of the sequential composition operator.

The process $\mathsf{t}(P,\, F,\, B,\, C)$ defines a transaction; in particular, P is the main process – the *body* –, to be executed in a transactional way; F and B are, respectively, the *failure manager* and the *failure bag*, to be executed if a failure occurs; and C is the *compensation*, to be executed in case an enclosing transaction fails. Only the main process, the failure manager, and the compensation are considered in BizTalk specifications. The further argument – the "failure bag" – is a repository to store the compensations of the enclosed transactions that complete while the enclosing transaction is still running. This repository has been added for semantic reasons, to describe the behaviour of (successfully) terminating transactions.

Process contexts are processes with a hole inside. Contexts are ranged over by $\mathbf{C}[\,]$ and are defined by the following grammar:

$$\begin{aligned}\mathbf{C}[\,] ::= {} & [\,] \\ & \mid \mathbf{C}[\,] \mid P \\ & \mid \mathbf{C}[\,]; P \\ & \mid (x)\mathbf{C}[\,] \\ & \mid \mathsf{t}(\mathbf{C}[\,],\, P,\, P,\, P)\end{aligned}$$

We use $\mathbf{C}[P]$ to denote the process obtained by substituting the hole inside $\mathbf{C}[\,]$ with the process P.

2.2 Structural Congruence

Structural congruence, written $\equiv$, equates all processes we will never want to distinguish for any semantic reason.

Definition 1. Structural congruence*, written $\equiv$, is the least congruence over processes, which contains α-renaming, and the following equations:*

$$\begin{array}{lll} P \mid Q \equiv Q \mid P & P \mid (Q \mid R) \equiv (P \mid Q) \mid R & \\ (P;Q);R \equiv P;(Q;R) & & \\ (x)(y)P \equiv (y)(x)P & (x)(P \mid Q) \equiv P \mid (x)Q & x \notin \mathsf{fv}(P) \\ (x)(P;Q) \equiv (x)P;Q \quad x \notin \mathsf{fv}(Q) & & \end{array}$$

$$(\overline{x}\langle\tilde{u}\rangle \mid P);Q \equiv \overline{x}\langle\tilde{u}\rangle \mid P;Q$$

$$\begin{array}{ll} \mathit{done} \mid P \equiv P & \mathit{abort} \mid \mathit{abort} \equiv \mathit{abort} \\ \mathit{done}; P \equiv P & \mathit{abort}; P \equiv \mathit{abort} \end{array}$$

$$K(\widetilde{v}) \equiv P\{^{\widetilde{v}}/_{\widetilde{u}}\} \qquad \textit{if } K(\widetilde{u}) \overset{def}{=} P$$

$$\begin{array}{ll} \mathsf{t}((x)P,\, F,\, B,\, C) \equiv (x)\mathsf{t}(P,\, F,\, B,\, C) & x \notin \mathsf{fv}(F) \cup \mathsf{fv}(B) \cup \mathsf{fv}(C) \\ \mathsf{t}(\overline{x}\langle\widetilde{u}\rangle \mid P,\, F,\, B,\, C) \equiv \overline{x}\langle\widetilde{u}\rangle \mid \mathsf{t}(P,\, F,\, B,\, C) & \\ (\mathsf{t}(\mathit{done},\, F,\, B,\, C) \mid P);P' \equiv \mathsf{t}(\mathit{done},\, F,\, B,\, C) \mid (P;P') & \end{array}$$

The first group of equations is almost standard: let us discuss the not standard ones. Notice that the rule $(x)(P;Q) \equiv (x)P;Q$ if $x \notin \mathsf{fv}(Q)$, is necessary (in combination with α–renaming) to allow restrictions to float at top level. The equation $(\overline{x}\langle\tilde{u}\rangle \mid P);Q \equiv \overline{x}\langle\tilde{u}\rangle \mid P;Q$ floats outputs outside sequences, since they have no continuation. Equations $\mathsf{done} \mid P \equiv P$ and $\mathsf{done}; P \equiv P$ specify that done is the identity of parallel and sequence; $\mathsf{abort} \mid \mathsf{abort} \equiv \mathsf{abort}$ states that a process is aborted when all its parallel components are aborted; $\mathsf{abort}; P \equiv \mathsf{abort}$ specifies that an aborted process is such, regardless of the continuation. The equation $\mathsf{t}(\overline{x}\langle\widetilde{u}\rangle \mid P,\ F,\ B,\ C) \equiv \overline{x}\langle\widetilde{u}\rangle \mid \mathsf{t}(P,\ F,\ B,\ C)$ allows to move outputs from inside a transaction to the outside environment and vice-versa. The intended semantics is the following. If a transactional process emits a message, this message traverses the transaction boundary, until reaching the corresponding input. In case the transaction fails, recoveries for this output may be detailed inside the processes F and B. The equation $(\mathsf{t}(\mathsf{done},\ F,\ B,\ C) \mid P); P' \equiv \mathsf{t}(\mathsf{done},\ F,\ B,\ C) \mid (P; P')$ allows to float successfully terminated transactions outside parallels and sequences. We observe that $\mathsf{t}(\mathsf{done},\ F,\ B,\ C)$ is not equal to done because, if an enclosing transaction fails, then the compensation C must be fired to accomodate possible inconsistencies (see the next rule (T-DONE)).

2.3 The Reduction Relation

The reduction relation of $\pi\mathsf{t}$-calculus is the least relation satisfying the rules in Table 1,

Table 1. The reduction rules of $\pi\mathsf{t}$-calculus

(RED)	$\overline{x}\langle\widetilde{v}\rangle \mid x(\widetilde{u}).P \ \rightarrow\ P\{^{\widetilde{v}}/_{\widetilde{u}}\}$
(T-DONE)	$\mathsf{t}(\mathsf{t}(\mathsf{done},\ F,\ B,\ C) \mid P,\ F',\ B',\ C') \ \rightarrow\ \mathsf{t}(P,\ F',\ B' \mid C,\ C')$
(T-ABORT)	$\mathsf{t}(\mathsf{abort},\ F,\ B,\ C) \rightarrow B; F$
(CONTEXT)	$\dfrac{P \rightarrow P'}{\mathbf{C}[P] \rightarrow \mathbf{C}[P']}$
(LIFT)	$\dfrac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q \equiv Q'}{P \rightarrow Q}$

The rules (T-DONE) and (T-ABORT) deserve some discussion. (T-DONE) models the successful completion of a transaction $\mathsf{t}(\mathsf{done},\ F,\ B,\ C)$. In this case, the compensation C must be recorded in the failure bag of the enclosing transaction, if any, to account for possible failures of the latter. If the outer transaction fails, rule (T-ABORT) specifies that the failure manager must be executed *after* the compensation of every enclosed transaction.

2.4 Comparison with the π-Calculus

It is interesting to observe that the $\pi\mathsf{t}$-calculus is essentially an extension of the π-calculus. Indeed, we can obtain the latter from the former simply by eliminating the sequence operator $P; Q$, the transactional process $\mathsf{t}(\mathsf{done},\ F,\ B,\ C)$, the process abort, and by interpreting the process done as the empty process $\mathsf{0}$ of the π-calculus. Actually it is also possible to encode the former into the latter, and we will study the encoding in section 4.

3 Examples

In the examples we use an extension of the calculus that comprises conditionals, boolean values, and boolean variables. Namely, we consider the operator:

$\mathsf{if}\ (a = k)\ \mathsf{then}\ P\ \mathsf{else}\ Q$

where $k = 0$ or $k = 1$, and a is a boolean variable. The semantics of the conditionals is the standard one.

3.1 Authentication

The first example describes a server that authenticates its clients exploiting a certification authority. The are four actors: the client (*Client*), the server (*Server*), the certification authority (*Auth*), and a law authority (*Law*) used to notify abuses.

We focus on the behaviour of the server: on reception of a request (which includes the identity of the client *id*, and its certificate *cert*), the server asks the certification authority to check the validity of the received certificate. Therefore, the server waits for an answer, that may be either *1* or *0*, depending on success or failure, respectively. In the case of success the server executes the required activity, on the contrary it communicates the abuse to the law authority. The following channels are used:

– *req* is the channel between *Client* and *Server*;
– *check* is the channel between *Server* and *Auth*;
– *resp* is a channel created by the *Main* process and passed to the *Server*: this channel is used to communicate the result (success or failure) of the certificate check (the same technique will be used also in the following examples);
– *ntf* is the channel between *Server* and *Law*;
– *abuse* is a local channel used to store data concerned with the abuse of certificates.

We are now in place to specify the above process in $\pi\mathsf{t}$-calculus:

$$\begin{array}{lll}
\mathit{Server} & = & (\mathit{abuse})\mathsf{t}(\mathit{Main},\ \mathit{Fail},\ \mathsf{done},\ \mathsf{done}) \\
\mathit{Main} & = & \mathit{req}(\mathit{task}, \mathit{id}, \mathit{cert}). \\
 & & \quad (\mathit{resp})(\overline{\mathit{check}}\langle \mathit{id}, \mathit{cert}, \mathit{resp}\rangle \mid \\
 & & \qquad \mathit{resp}(a).\mathsf{if}\ (a = 1)\ \mathsf{then}\ \mathit{Execute}(\mathit{task})\ \mathsf{else}\ (\overline{\mathit{abuse}}\langle \mathit{id}, \mathit{cert}\rangle \mid \mathsf{abort}) \\
 & & \qquad) \\
\mathit{Fail} & = & \mathit{abuse}(\mathit{id}, \mathit{cert}).\overline{\mathit{ntf}}\langle \mathit{id}, \mathit{cert}\rangle
\end{array}$$

where *Execute* is a program constant (that we leave unspecified) representing the execution of the task. It is important to note that the transaction is used here merely as a facility for exception management.

3.2 Flight or Train Booking

This second example has been introduced to discuss that, when an abort process is reached, the failure process is not activated immediately, but the termination of parallel threads is waited. This is particularly useful in transactions, composed by concurrent activities, in order to give a chance to any activity to terminate successfully its task even if some other fails.

We consider a travel agency (*Travel*) that books a certain number of flights: each reservation is managed by a process *Reserve*. The reservations may succeed or fail. At the end of all the reservation subtasks, if at least one of them has failed, a failure process is started which reserves trains instead of failed flight reservations.

We exploit the following names:

- *bookF* is a channel shared between the travel agency and the flight company;
- *dest* and *resp* are names fresh for each process *Reserve*: *dest* indicates the destination while *resp* is the channel to be used to indicate the success or failure of the reservation (using *1* or *0*, respectively);
- *train* is a channel used to store, in the case of flight reservation failure, the request for an alternative train reservation;
- *bookT* is a channel shared between the travel agency and the train company.

We are now in place to specify the booking process:

$$\begin{aligned}
Travel &= \mathsf{t}(Flight,\ Train,\ \mathsf{done},\ \mathsf{done})\\
Flight &= Reserve \mid Reserve \mid \ldots \mid Reserve\\
Reserve &= (dest)(resp)(\ \overline{bookF}\langle dest, resp\rangle \mid\\
&\qquad\qquad resp(a).\mathsf{if}\ (a = 1)\ \mathsf{then\ done\ else}\ (\overline{train}\langle dest\rangle \mid \mathsf{abort})\ )\\
Train &= train(dest).(\overline{bookT}\langle dest\rangle \mid Train)
\end{aligned}$$

As described above, the failure of one of the *Reserve* processes does not influence the concurrent activities; the failure process starts only on termination of all the *Reserve* processes. This is ensured by the rule (T-ABORT) that activates the failure bag and the failure manager only when the main process is (structurally congruent to) the process abort.

3.3 Flight and Hotel Booking

In this third example we decribe a transaction (*Journey*) composed of the sequence of two transactions: the first books a return flight ticket, the second reserves a hotel room for the nights between the arrival and the departure dates.

The first transaction has a compensation process which is responsible for cancelling the flight reservation. We use the two process constants *Ticket(dest,a,d)* and *Room(dest,a,d)* to represent the two transactions: *dest* is the destination while *a* and *d* are the arrival and departure dates, respectively. We consider the existence of three channels:

- *bookF* and *bookH* used to ask for a flight or a room booking, respectively. In both cases, the request may either succeed or fail;
- *cancelF* is used to ask for the cancellation of a flight reservation.

The booking requests may either succed or fail: also in this case we use the boolean values *1* and *0* to denote these two possible outcomes, respectively.

We are now in place to present the formal specification of the *Journey* process:

$$\begin{array}{lll}
\mathit{Journey} & = \mathsf{t}(\,\mathit{Ticket}(\mathit{dest},a,d);\mathit{Room}(\mathit{dest},a,d),\ \mathsf{done},\ \mathsf{done},\ \mathsf{done}) \\
\mathit{Ticket}(\mathit{dest},a,d) & = (\mathit{resp})\ \mathsf{t}(\ \overline{\mathit{bookF}}\langle \mathit{dest},a,d,\mathit{resp}\rangle \\
 & \qquad\qquad\quad \mid \mathit{resp}(\mathit{ack}).\mathsf{if}\ (\mathit{ack}=1)\ \mathsf{then\ done\ else\ abort}, \\
 & \qquad\qquad\quad \mathsf{done}, \mathsf{done}, \overline{\mathit{cancelF}}\langle \mathit{dest},a,d\rangle\ ) \\
\mathit{Room}(\mathit{dest},a,d) & = (\mathit{resp})\ \mathsf{t}(\ \overline{\mathit{bookH}}\langle \mathit{dest},a,d,\mathit{resp}\rangle \\
 & \qquad\qquad\quad \mid \mathit{resp}(\mathit{ack}).\mathsf{if}\ (\mathit{ack}=1)\ \mathsf{then\ done\ else\ abort}, \\
 & \qquad\qquad\quad \mathsf{done}, \mathsf{done}, \mathsf{done}\ )
\end{array}$$

Notice that an equivalent behaviour is obtained placing the flight cancellation request $\overline{\mathit{cancelF}}\langle \mathit{dest},a,d\rangle$ as failure of the second transaction. However, the process specification we have reported supports better modularity because all the activities related to flight reservation, as well as cancellation, are all inside the same transaction.

4 The Encoding of the πt-Calculus into the Asynchronous π-Calculus

In this section we demonstrate that πt-calculus may be encoded into the asynchronous π-calculus. We recall that the latter is indeed a subcalculus of the former (see section 2.4), therefore we avoid any redefinition. The encoding is a partial function $[\![-]\!]^{-;-}_{-;-}$. The process $[\![P]\!]^{z;w}_{x;y}$, such that $x,y,z,w \notin \mathsf{fv}(P)$, uses the channels x, y, z and w respectively to signal successful termination without compensations, erroneous termination without compensations, successful termination with compensation and erroneous termination with compensation. The termination of P could have compensations if there are transactional contexts defined in it.

In the definition below we use the constants M and N^{zw}_{xy} defined as follows:

$$\begin{array}{l}
M(c,c',c'') = c(x,y,z,w).\ (x_L,y_L,z_L,w_L,x_R,y_R,z_R,w_R)(\\
\qquad\qquad \overline{c'}\langle x_L y_L z_L w_L\rangle \mid \overline{c''}\langle x_R y_R z_R w_R\rangle \\
\qquad\qquad \mid N^{zw}_{xy}(x_L,y_L,z_L,w_L,x_R,y_R,z_R,w_R) \\
\qquad)
\end{array}$$

$$\begin{array}{l}
N^{zw}_{xy}(x_L,y_L,z_L,w_L,x_R,y_R,z_R,w_R) = \\
\quad x_L().(x_R().\overline{x}\langle\rangle \mid y_R().\overline{y}\langle\rangle \mid z_R(c_R).\overline{z}\langle c_R\rangle \mid w_R(c_R).\overline{w}\langle c_R\rangle) \\
\quad \mid y_L().(x_R().\overline{y}\langle\rangle \mid y_R().\overline{y}\langle\rangle \mid z_R(c_R).\overline{w}\langle c_R\rangle \mid w_R(c_R).\overline{w}\langle c_R\rangle) \\
\quad \mid z_L(c_L).(\ x_R().\overline{z}\langle c_L\rangle \mid y_R().\overline{w}\langle c_L\rangle \\
\qquad\qquad \mid z_R(c_R).(c''')(\overline{z}\langle c'''\rangle \mid M(c''',c_L,c_R)) \\
\qquad\qquad \mid w_R(c_R).(c''')(\overline{w}\langle c'''\rangle \mid M(c''',c_L,c_R))) \\
\quad \mid w_L(c_L).(x_R().\overline{w}\langle c_R\rangle \mid y_R().\overline{w}\langle c_R\rangle \\
\qquad\qquad \mid z_R(c_R).(c''')(\overline{w}\langle c'''\rangle \mid M(c''',c_L,c_R)) \\
\qquad\qquad \mid w_R(c_R).(c''')(\overline{w}\langle c'''\rangle \mid M(c''',c_L,c_R)))
\end{array}$$

The purpose of M and N_{xy}^{zw} is discussed next. Actually we focus on M, because N_{xy}^{zw} is similar. The process $M(c, c', c'')$ multiplexes the compensation request coming from c towards c' and c'', which are the channels for invoking compensations of two parallel subprocesses, let us call them L and R. The subprocesses L and R may return four different kinds of signals, namely x_L, y_L, z_L, w_L and x_R, y_R, z_R, w_R, with sixteen different combinations. The interesting combinations are when L returns on z_L or w_L, and R returns on z_R or w_R, namely when L and R return a compensation to activate in case of failure. In these cases a new multiplexer must be triggered, henceforth the recursive invocation of M. We remark that, for the correctness of M and N, it is necessary there are exactly two messages: one on channels x_L, y_L, z_L, w_L and the other on x_R, y_R, z_R, w_R.

Definition 2. *The process $[\![P]\!]_{x,y}^{z,w}$, such that $x, y, z, w \notin \mathsf{fv}(P)$, is defined by the equations below (the definition of M_P is the last one). We always assume that new names introduced by the encoding never clash with free names of the encoded process.*

$$[\![\mathsf{done}]\!]_{x,y}^{z,w} = \overline{x}\langle\rangle \qquad [\![\mathsf{abort}]\!]_{x,y}^{z,w} = \overline{y}\langle\rangle$$

$$[\![\overline{u}\langle\widetilde{v}\rangle]\!]_{x,y}^{z,w} = \overline{u}\langle\widetilde{v}\rangle \mid \overline{x}\langle\rangle \qquad [\![u(\widetilde{v}).P]\!]_{x,y}^{z,w} = u(\widetilde{v}).[\![P]\!]_{x,y}^{z,w} \qquad (x, y, z, w \notin \widetilde{v})$$

$$[\![(u)P]\!]_{x,y}^{z,w} = (u)[\![P]\!]_{x,y}^{z,w} \qquad (u \notin x, y, z, w)$$

$$\begin{aligned}
[\![P; Q]\!]_{x,y}^{z,w} = (x', y', z', w')(\; & [\![P]\!]_{x',y'}^{z',w'} \\
& \mid x'().[\![Q]\!]_{x,y}^{z,w} \\
& \mid y'().\overline{y}\langle\rangle \\
& \mid z'(c).M_Q(c, z, w) \\
& \mid w'(c).\overline{w}\langle c\rangle \\
)&
\end{aligned}$$

$$\begin{aligned}
[\![P \mid Q]\!]_{x,y}^{z,w} = (x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R)(& \\
& [\![P]\!]_{x_L,y_L}^{z_L,w_L} \mid [\![Q]\!]_{x_R,y_R}^{z_R,w_R} \\
& \mid N_{xy}^{zw}(x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R) \\
)&
\end{aligned}$$

$$\begin{aligned}
[\![\mathsf{t}(P,\ F,\ B,\ C)]\!]_{x,y}^{z,w} = (x_1, y_1, z_1, w_1)(& \\
& [\![P]\!]_{x_1,y_1}^{z_1,w_1} \\
& \mid x_1().(c)(\overline{z}\langle c\rangle \mid c(x_1 y_1 z_1 w_1).[\![C]\!]_{x_1,y_1}^{z_1,w_1}) \\
& \mid y_1().[\![B; F]\!]_{x,y}^{z,w} \\
& \mid z_1(c).(c')(\overline{z}\langle c'\rangle \mid c'(x_1 y_1 z_1 w_1).[\![C]\!]_{x_1,y_1}^{z_1,w_1}) \\
& \mid w_1(c).(x', y', z', w') \\
& \qquad (x_L, y_L, z_L, w_L) \\
& \qquad (x_R, y_R, z_R, w_R)(\\
& \qquad\quad \overline{c}\langle x_L y_L z_L w_L\rangle \mid [\![B]\!]_{x_R,y_R}^{z_R,w_R} \\
& \qquad\quad \mid N_{x'y'}^{z'w'}(x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R) \\
& \qquad\quad \mid x'().[\![F]\!]_{x,y}^{z,w} \\
& \qquad\quad \mid y'().\overline{y}\langle\rangle \\
& \qquad\quad \mid z'(c').M_F(c', z, w) \\
& \qquad\quad \mid w'(c').\overline{w}\langle c'\rangle \\
& \qquad) \\
)&
\end{aligned}$$

$$[\![K(u_1,\ldots,u_n)]\!]^{z,w}_{x,y} = K_\pi(u_1,\ldots,u_n,x,y,z,w)$$

$$K_\pi(v_1,\ldots,v_n,x,y,z,w) = [\![P]\!]^{z,w}_{x,y} \qquad (\textit{assuming } \; K(v_1,\ldots,v_n) \overset{def}{=} P)$$

where the definition of the constant M_P is the following:

$$\begin{array}{rl} M_P(c,z,w) = (x',y',z',w')(& [\![P]\!]^{z',w'}_{x',y'} \\ & \mid x'().\overline{z}\langle c\rangle \\ & \mid y'().\overline{w}\langle c\rangle \\ & \mid z'(c').(c'')(\overline{z}\langle c''\rangle \mid M(c'',c,c')) \\ & \mid w'(c').(c'')(\overline{w}\langle c''\rangle \mid M(c'',c,c')) \\ ) & \end{array}$$

Let us clarify the behaviour of $[\![\cdot]\!]^{z,w}_{x,y}$, according to the shape of the argument.

If the argument is done, a signal on x is emitted, meaning the successful termination without compensations.

If the argument is $\overline{u}\langle\widetilde{v}\rangle$, the successful termination signal is in addition with the message on u.

If the argument is abort, a signal on y is emitted, representing the failure without compensation.

If the argument is $u(\tilde{v}).Q$ or $(x)Q$ the encoding is homomorphic and successes and failures are passed to the encoding of Q.

If the argument is $P;Q$, the encoding of P is executed and, in case of successful completion, the encoding of Q is performed afterwards (we observe that the encoding of Q is always underneath an input). In case there are compensations, the process M_Q is called, as discussed above.

If the argument is $P \mid Q$, the encodings of P and Q are performed in parallel and the results are collected by the agent N.

If the argument is $\mathsf{t}(P,\ F,\ B,\ C)$, the encoding of the body P is executed. There are several cases. In case of success (with or without compensations), the process $[\![\mathsf{t}(P,\ F,\ \mathsf{done},\ C)]\!]^{z,w}_{x,y}$ emits on z the compensation triggering the encoding of C (inner compensations are discarded). In case of failure without compensations, we must perform the encoding of the agent B *before* the failure manager F. In case of failure with compensations then the failure manager F must be executed *after* the other compensations. Remark that this last case is very similar to the sequential composition.

If the argument is $K(u_1,\ldots,u_n)$, we use a twin constant K_π that carries four additional arguments, namely the channels for signaling success and failure. The definition of K_π is the expected encoding of the definition of K.

4.1 Correctness of the Encoding

We assess the correctness of the encoding of $\pi\mathsf{t}$-calculus with respect to the semantics defined in section 2. Since the previous encoding yields π-calculus agents that show up several deadlocked subprocesses (see the definitions of sequence, parallel, or transaction), we require an extensional semantics that is, as far as the π-calculus is considered, insensitive to deadlocked processes. To this aim we introduce *(weak) barbed equivalence* [17].

Definition 3. *Let $P \downarrow x$ be the least relation satisfying the rules below.*

$$\begin{array}{ll}
\overline{x}\langle\widetilde{u}\rangle \downarrow x & \\
P \mid Q \downarrow x & \textit{if } P \downarrow x \textit{ or } Q \downarrow x \\
P; Q \downarrow x & \textit{if } P \downarrow x \\
(y)P \downarrow x & \textit{if } P \downarrow x \textit{ and } x \neq y \\
\mathsf{t}(P, F, B, C) \downarrow x & \textit{if } P \downarrow x \\
P \downarrow x & \textit{if } Q \downarrow x \textit{ and } P \equiv Q
\end{array}$$

If $P \downarrow x$ we say that P has a barb *on x.*

Notice that, this definition of barb is different from the standard one [17] because we are closing the relation by structural equivalence. The usual definition by induction on the syntax is hard in our case because of the sequence operator. We remark that the above barb does not allow to discriminate between the processes done and abort, even though a transaction context behaves differently when filled with them. Actually this is not an issue since the barbed congruence semantics (not discussed in this paper) is enough discriminating to separate done and abort.

Definition 4. *A* (weak) barbed bisimulation *is a symmetric binary relation $\mathcal{S}$ between agents such that $P \mathrel{\mathcal{S}} Q$ implies:*

1. *If $P \to P'$ then $Q \to^* Q'$ and $P' \mathrel{\mathcal{S}} Q'$.*
2. *If $P \downarrow x$ for some x, then $Q \to^* Q'$ and $Q' \downarrow x$.*

P is barbed bisimilar to Q, written $P \stackrel{\bullet}{\approx} Q$, if there exists some barbed bisimulation $\mathcal{S}$ such that $P \mathrel{\mathcal{S}} Q$.

For instance, done $\stackrel{\bullet}{\approx}$ abort $\stackrel{\bullet}{\approx}$ $(x)\overline{x}\langle u\rangle$. The definitions of barb and barbed equivalence coincide with those of π-calculus when processes are restricted to π-calculus ones.

The correctness of the encoding $[\![P]\!]^{z,w}_{x,y}$ is formalized by the following result.

Theorem 1. *Let P be a $\pi\mathsf{t}$-calculus process. Then*

1. *$P \downarrow u$ if and only if $[\![P]\!]^{z,w}_{x,y} \downarrow u$ and $u \notin \{x, y, z, w\}$.*
2. *If $P \to Q$ then $[\![P]\!]^{z,w}_{x,y} \to^* \stackrel{\bullet}{\approx} [\![Q]\!]^{z,w}_{x,y}$ (provided that x, y, z, w do not clash with $\mathsf{fv}(P)$ and $\mathsf{fv}(Q)$).*
3. *If $[\![P]\!]^{z,w}_{x,y} \to Q$ then there is R such that $P \to^* R$ and $Q \stackrel{\bullet}{\approx} [\![R]\!]^{z,w}_{x,y}$.*

Proof. (Sketch) (1) Since "$\downarrow$" encompasses structural equivalence, one ends up at demonstrating that, if $P \equiv Q$ and $[\![P]\!]^{z,w}_{x,y} \downarrow u$, then $[\![Q]\!]^{z,w}_{x,y} \downarrow u$. This is mostly a straightforward analysis, except for the structural rules $(\overline{x}\langle\tilde{u}\rangle \mid P); Q \equiv \overline{x}\langle\tilde{u}\rangle \mid P; Q$ and $(\mathsf{t}(\mathsf{done}, F, B, C) \mid P); P' \equiv \mathsf{t}(\mathsf{done}, F, B, C) \mid (P; P')$. Both cases follow by the definition of the encoding and by a careful analysis whether P is amenable to done or not.

(2) We analyze the basic reductions. Among them, the difficult case is (T-DONE) because of the management of the failure bag. Let us discuss this case in detail. On one side we have:

$$\mathsf{t}(\mathsf{t}(\mathsf{done}, F, B, C) \mid P, F', B', C') \;\to\; \mathsf{t}(P, F', B' \mid C, C')$$

On the other side we have:

$$
\begin{array}{l}
[\![\mathsf{t}(\mathsf{t}(\mathsf{done},\ F,\ B,\ C) \mid P,\ F',\ B',\ C')]\!]^{z,w}_{x,y} \\
\quad = (x_1, y_1, z_1, w_1)([\![\mathsf{t}(\mathsf{done},\ F,\ B,\ C) \mid P]\!]^{z_1,w_1}_{x_1,y_1} \mid T^{xyzw}_{x_1y_1z_1w_1}(F',B',C')) \\
\qquad\qquad\qquad\qquad\qquad\quad (T^{xyzw}_{x_1y_1z_1w_1}(F',B',C') \text{ is the manager of transaction} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{that may be grabbed from definition 2)} \\[1ex]
\quad = (x_1, y_1, z_1, w_1)(x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R)(\\
\qquad [\![\mathsf{t}(\mathsf{done},\ F,\ B,\ C)]\!]^{z_L,w_L}_{x_L,y_L} \mid [\![P]\!]^{z_R,w_R}_{x_R,y_R} \mid N^{z_1,w_1}_{x_1,y_1} \\
\qquad (x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R) \mid T^{xyzw}_{x_1y_1z_1w_1}(F',B',C')) \\[1ex]
\quad \rightarrow (x_1, y_1, z_1, w_1)(x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R)(\\
\qquad (c)(\overline{z_L}\langle c\rangle \mid c(x'y'z'w').[\![C]\!]^{z',w'}_{x',y'} \mid [\![P]\!]^{z_R,w_R}_{x_R,y_R} \\
\qquad \mid N^{z_1,w_1}_{x_1,y_1}(x_L, y_L, z_L, w_L, x_R, y_R, z_R, w_R) \mid T^{xyzw}_{x_1y_1z_1w_1}(F',B',C')) \\[1ex]
\quad \rightarrow (x_R, y_R, z_R, w_R)([\![P]\!]^{z_R,w_R}_{x_R,y_R} \mid \\
\qquad\quad (x_1, y_1, z_1, w_1)(c)(x_R().\overline{z_1}\langle c\rangle \mid y_R().\overline{w_1}\langle c\rangle \\
\qquad\qquad \mid z_R(c_R).(c''')(\overline{z_1}\langle c'''\rangle \mid M(c''', c, c_R)) \\
\qquad\qquad \mid w_R(c_R).(c''')(\overline{w_1}\langle c'''\rangle \mid M(c''', c, c_R)) \\
\qquad\qquad \mid c(x'y'z'w').[\![C]\!]^{z',w'}_{x',y'} \mid T^{xyzw}_{x_1y_1z_1w_1}(F',B',C')))
\end{array}
$$

Therefore we are reduced to prove that

$$
\begin{array}{l}
(x_1, y_1, z_1, w_1)(c)(x_R().\overline{z_1}\langle c\rangle \mid y_R().\overline{w_1}\langle c\rangle \\
\quad \mid z_R(c_R).(c''')(\overline{z_1}\langle c'''\rangle \mid M(c''', c, c_R)) \\
\quad \mid w_R(c_R).(c''')(\overline{w_1}\langle c'''\rangle \mid M(c''', c, c_R)) \\
\quad \mid c(x'y'z'w').[\![C]\!]^{z',w'}_{x',y'} \mid T^{xyzw}_{x_1y_1z_1w_1}(F',B',C'))
\end{array}
\quad \dot{\approx}\ T^{xyzw}_{x_Ry_Rz_Rw_R}(F', B' \mid C, C')
$$

This follows by a close inspection of all the possible cases.

(3) The proof consists of picking some representative π-calculus processes of the evaluation of $[\![P]\!]$, and demonstrating that intermediate processes are barbed bisimilar to the representatives. Representatives are processes where no "bureaucratic reactions" is possible (these are the reactions due to the encoding). Representatives are proved bisimilar to encodings of $\pi\mathsf{t}$-calculus processes in a way similar to that reported for the case (2).

This theorem is the basic result to relate barbed bisimulation in π-calculus and in $\pi\mathsf{t}$-calculus. It is well-known that this equivalence is not very interesting because its discriminating power is weak. Nevertheless, our intended application of such result is to infer barbed bisimulation congruence of $\pi\mathsf{t}$-calculus – which, on the contrary, is an interesting semantics – from barbed bisimulation congruence of the encoded agents in π-calculus. This is a considerable result for our calculus that requires a weighty effort (e.g. we should exploit an equivalent labelled semantics characterizations [17]) that we leave for future work.

5 Conclusions

Long-running transactions have recently received a renewed interest with the advent of Web Services-based business interactions. Indeed, these transactions are considered a valuable tool for business process modeling. In this paper, we have formalized and studied the notion of long-running transactions incorporated in Microsoft BizTalk [14], a visual environment for business process modeling.

We notice the absence, to the best of our knowledge, of formal specifications and analysis of transactions in Web Services-based business process modeling. The unique published work we are aware of is [5], devoted to the investigation of ACID (short-lived) transactions in the context of BizTalk.

As future work, we plan to investigate more complex mechanisms for composing transactions; in particular, in our calculus a transaction obtained as parallel composition of sub-transactions waits for the termination of all these transactions before terminating itself. Other interesting composition operators, see e.g. the *pick* constructur of XLANG [18], allows for the execution of one subcomponent only, chosen according to the occurrence of some specific event.

Acknowledgement

We thank the anonymous referees for their fruitful comments and Greg Meredith and Tony Hoare for the insightful discussions about the topics treated in this paper. This paper has been partially supported by Italian Ministry of University – MURST 40% – Progetto NAPOLI, and by a grant by Microsoft Research.

References

1. A. Ankolekar, M. Bursten, J. Hobbs, O. Lassila, D. Martin, S. McIlraith, S. Narayanan, N. Paolucci, T. Payne, K. Sycara, H. Zeng, (2001). DAML-S: Semantic Markup for Web Services. In International Semantic Web Working Symposium 2001.
2. J.C.M. Baeten and W.P. Weijland. Process algebra. Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press 1990.
3. T. Berners-Lee, D. Brickley, D. Connolly, M. Dean, S. Decker, D. Fensel, R. Fikes, P. Hayes, J. Heflin, J. Hendler, O. Lassila, D. McGuinness, L.A. Stein. DAML+OIL. [http://www.daml.org/2001/03/daml+oil-index], 2001.
4. G. Boudol. Asynchrony and the π-calculus. Technical Report 1702, INRIA, Sophia–Antipolis,1992.
5. R. Bruni, C. Laneve, U. Montanari. Orchestrating Transactions in Join Calculus. In proc. of CONCUR'02, LNCS 2421, pp. 321 - 337, 2002.
6. Business Process Modeling Language (BPML). [www.bpmi.org]
7. E. Christensen, F. Curbera, G. Meredith and S. Weerawarana. Web Services Description Language (WSDL 1.1). [www.w3.org/TR/wdsl], W3C, Note 15, 2001.
8. T.D.S. Coalition. DAML-S: Web service description for the semantic web. In Proc. of ISWC, 2002.

9. F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte and S. Weerawarana. Business Process Execution Language for Web Services (BPEL4WS 1.0). [http://www-106.ibm.com/developerworks/webservices/library/ ws-bpel/], 2002.
10. S. Dalal, S. Temel, M. Little, M. Potts, J, Webber. Coordinating Business Transactions on the Web. IEEE Internet Computing, January-February 2003.
11. H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, K. Salem. Modeling Long-Running Activities as Nested Sagas. IEEE Bulletin of the Technical Committee on Data Engineering, 14 (1), 1991.
12. H. Garcia-Molina, K. Salem. Sagas. In Proc. of SIGMOD International Conference on Management of Data, pp. 249–259, 1987.
13. F. Leymann. Web Services Flow Language (WSFL 1.0). [http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf], Member IBM Academy of Technology, IBM Software Group, 2001.
14. Microsoft BizTalk Server. [http://www.microsoft.com/biztalk/default.asp], Microsoft Corporation.
15. R. Milner, J. Parrow, D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77. Academic Press, 1992.
16. J. Roberts, K. Srinivasan. Tentative Hold Protocol Part 1: White Paper. W3C Note 28 November 20001. [http://www.w3.org/TR/tenthold-1/]
17. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*, Cambridge University Press, 2001.
18. S. Thatte. XLANG: Web Services for Business Process Design. [http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm], Microsoft Corporation, 2001.

Formal Analysis of Some Timed Security Properties in Wireless Protocols[*]

Roberto Gorrieri[1], Fabio Martinelli[2], Marinella Petrocchi[2], and Anna Vaccarelli[2]

[1] Dipartimento di Scienze dell'Informazione, Università di Bologna
gorrieri@cs.unibo.it
[2] Istituto di Informatica e Telematica – C.N.R.
{fabio.martinelli,marinella.petrocchi,anna.vaccarelli}@iit.cnr.it

Abstract. We show how a recent language for the description of cryptographic protocols in a real time setting may be suitable to formally verify security aspects of wireless protocols. We define also a compositional proof rule for establishing security properties of such protocols. The effectiveness of our approach is shown by defining and studying the timed integrity property for μTESLA, a well-known protocol for wireless sensor networks. We are able to deal with protocol specifications with an arbitrary number of agents (senders as well as receivers) running the protocol.

Keywords: Security, Wireless Communication, Formal Analysis, Sensor Networks.

1 Introduction

Ambient intelligence is one of the main research issues in Europe. It imposes a view of the world where objects may interact with each other by means of wireless communications. In our framework, we are interested in developing formal analysis techniques for such scenarios. In particular, we aim at studying security properties of wireless networks and we start with the analysis of a secure wireless protocol for sensor networks where time plays an essential role.

Wireless systems consisting of mobile nodes self-organizing in temporary routing topologies form wireless *ad hoc* networks. Wireless *ad hoc* networks may cover various types of applications. Peculiarity in their usage is when wired infrastructure is not available at all (a typical example involves rescue operations in remote areas or disaster recovery operations). Various issues in the world of the *ad hoc* networks are greatly influenced by temporal relationships (e.g., whichever

[*] Work partially supported by MURST Project "Metodi Formali per la Sicurezza ed il Tempo" (MEFISTO); by MIUR project COVER; by IST-FET project "Design Environments for Global ApplicationS (DEGAS)"; by CNR project "Tecniche e Strumenti Software per l'Analisi della Sicurezza delle Comunicazioni in Applicazioni Telematiche di Interesse Economico e Sociale" and by a CSP grant for the project "SeTAPS II".

the media access control protocol may be, real-time and temporal synchronization constraints characterize the hosts' communications). The reader can refer to [14] for a full-detailed survey about real-time issues in *ad hoc* networks. Actually, we are mainly interested in time synchronization issues for a subclass of *ad hoc* networks, i.e. wireless sensor networks. A wireless sensor network is typically composed of hundreds or thousands of sensors, (up to) cubic millimeters devices provided with autonomous sensing, computation and communication. The network is coordinated in a distributed mode in order to collect information on their surroundings. Sensor networks applications are expected to range over many fields, from home applications like automation and smart environments to military uses (monitoring equipment and ammunitions, battlefield management, etc.). Sensors may be used in a building for heating and air conditioning control as well as in a hospital for medical monitoring (e.g. drugs administration or telemonitoring of physiological conditions of the patients), not to mention environmental monitoring, such as the detection of possible fires in a forest.

Researchers have recently addressed the quest for securing wireless sensor networks communication. Sensors already have to cope with severe constraints in terms of power consumption, bandwidth, storage and may not have the resources to perform cryptographic operations in their completeness. Standard solutions developed for conventional computers cannot be applied, hence new schemes have been proposed and surveys have been carried out (e.g. [10, 13]). Temporal constraints occur in [13], where a time synchronization is required between a base station and the sensors in the network.

In [8] it was developed a compositional analysis technique able to deal with multicast/broadcast protocols. Actually, in that first work we were not able to manage protocols with time-dependent security properties. In this paper we aim at enhancing that analysis technique in order to cope with wireless protocols where a time synchronization is required, e.g. [13]. Among the properties we are now able to check are timed secrecy and timed integrity. The former requires that a message is kept secret only for a certain amount of time and the latter that a message sent in a time interval has not been altered during the communication. We use the approach based on non-interference developed in [5] as a method to express security properties.

This paper improves the work in [8] as follows: 1) a formal framework for modeling wireless communication protocols is outlined by means of a real-time process algebra; 2) a new compositional proof rule is given for dealing with timed security properties as timed secrecy and timed integrity; 3) a formal specification of μTESLA (see [13]) is defined and analyzed; it is checked that such a protocol enjoys the timed integrity property; 4) the analysis of μTESLA is carried out taking into consideration an arbitrary (but finite) number of senders and receivers. The previous work on a related protocol (the TESLA protocol, [12]), deals only with a fixed and small number of senders and receivers, [2].

This paper is organized as follows. Section 2 recalls the formal language we are going to adopt for the description of μTESLA. Section 3 presents the compositional analysis techniques in a timed setting. Section 4 introduces the μTESLA formal specifications. Section 5 shows how to apply the previous theories to the analysis of μTESLA. Finally, Section 6 concludes the paper.

2 A Real-Time Language for Cryptographic Protocols

The real-time extension of the *Cryptographic Security Process Algebra* (*CryptoSPA* for short) in [3, 5] has been proposed in [6]. The new language, *timedCryptoSPA* (*tCryptoSPA* for short), is adopted for describing cryptographic protocols where information about the concrete timing of events is necessary. We remind the reader of the syntax, the operational semantics of the language and some auxiliary notions.

The Syntax. The syntax of *tCryptoSPA* is based on a set $\mathcal{I}$ of input channels (ranged over by c), a set $\mathcal{O}$ of output channels (ranged over by $\overline{c}$), a set of closed terms $\mathcal{M}$ (a set of closed terms of a term algebra which, at least, contains encryption $\{m\}_k$ and pairing (m, m_1) operations), *Const* of constants, ranged over by A, and Var of variables, ranged over by x. The set $\mathcal{L}$ of *tCryptoSPA* processes is defined as:

$$P ::= \mathbf{0} \mid c(x).P \mid \overline{c}e.P \mid \tau.P \mid tick.P \mid P_1 + P_2 \mid P_1 \,|\, P_2 \mid P \backslash L \mid$$

$$A(e_1, \dots, e_n) \mid [\langle e_1, \dots, e_r \rangle \vdash_{rule} x] P_1 ; P_2$$

where $e, e_1, \dots, e_r, e_n$ are messages or variables and L is a set of channels. Both the operators $c(x).P$ and $[\langle e_1 \dots e_r \rangle \vdash_{rule} x] P_1 ; P_2$ bind the variable x in P, P_1.

Let $Def : Const \longrightarrow \mathcal{L}$ be a set of defining equations of the form $A(x_1, \dots, x_n) \doteq P$, where P may contain no free variables except $x_1, \dots, x_n$, which must be distinct. Constants permit us to define recursive processes, but we have to be a bit careful in using them. A term P is *closed* w.r.t. Def if all the constants occurring in P are defined in Def (and, recursively, for their defining terms). A term P is *guarded* w.r.t. Def if all the constants occurring in P (and, recursively, for their defining terms) occur in a prefix context [11].

The set *Act* of actions which may be performed by a system is defined as: $Act = \{c(m), \overline{c}m, \tau, tick, \mid c \in \mathcal{I}, \overline{c} \in \mathcal{O}, m \in \mathcal{M}, m \text{ closed}\}$. τ is the internal, invisible action. *tick* is the special action used to model time elapsing. We let l range over $Act \backslash \{tick\}$. We call $\mathcal{P}$ the set of all the *tCryptoSPA closed* terms (i.e., with no free variables) that are closed and guarded w.r.t. Def. We define *sort(P)* to be the set of all the channels syntactically occurring in the term P.

The informal semantics of the *tCryptoSPA* processes is the following:

- $\mathbf{0}$ is the process that does nothing;
- $c(x).P$ is the process that can receive a message m on channel c and then behaves like P. The received message replaces the variable x;
- $\overline{c}m.P$ is the process that can send m on channel c, then behaving like P;
- $\tau.P$ is the process that executes the internal, invisible action τ and then behaves like P;

- $tick.P$ is a process willing to let one time unit pass and then behaving as P;
- $P_1 + P_2$ (*choice*) represents the nondeterministic choice between the two processes P_1 and P_2; with respect to *tick* actions, time passes when both P_1 and P_2 are able to perform a *tick* action – and in such a case by performing *tick* a configuration where both the derivatives of the summands can still be chosen is reached. When only one of the two processes can perform *tick*, say P_1, it could be either that P_1 performs *tick* – and in such a case P_2 is discarded – or P_2 performs its normal activity – and in such a case P_1 is discarded; moreover, τ prefixed summands have priority over *tick* prefixed summands;
- $P_1 \,|\, P_2$ (*parallel*) is the parallel composition of processes that can proceed in an asynchronous way but they must synchronize on complementary actions to make a communication, represented by a τ. Both components must agree on performing a *tick* action, and this can be done even if a communication is possible (we do not have *maximal progress* assumption);
- $P \backslash L$ allows only visible actions whose channels are not in L;
- $A(m_1, \ldots, m_n)$ behaves like the respective defining term P where all the variables $x_1, \ldots, x_n$ are replaced by the messages $m_1, \ldots, m_n$;
- $[\langle e_1, \ldots, e_r \rangle \vdash_{rule} x]P_1; P_2$ is the process used to model message handling and cryptography. The process $[\langle e_1, \ldots, e_r \rangle \vdash_{rule} x]P_1; P_2$ tries to deduce an information z from the tuple of messages $\langle e_1, \ldots, e_r \rangle$ through the application of rule $\vdash_{rule}$; if it succeeds then it behaves like $P_1[z/x]$, otherwise like P_2. The set of rules that can be applied is defined through an inference system (e.g., see Figure 1).

Auxiliary Notions. The time model adopted in the language is known as the *fictitious clock* approach of, e.g., [9]. A global clock is supposed to be updated whenever all the processes agree on this, by globally synchronizing on the special action *tick*, representing the passing of a time unit. All the other actions are assumed to take no time.

In order to model message handling and cryptography we use a set of inference rules. Note that *tCryptoSPA* syntax, its semantics and the results obtained are completely parametric with respect to the inference system used. We show in Figure 1 a suitable inference system we are going to use in the following sections. This inference system can combine two messages obtaining a pair (rule $\vdash_{pair}$); it can extract one message from a pair (rules $\vdash_{fst}$ and $\vdash_{snd}$); it can apply a one-way hash function F to message x and obtain digest $F(x)$ (rule $\vdash_{hash}$) and finally compute the message authentication code (MAC) of a message with a key (rule $\vdash_{mac}$).

Given an inference system, we can define a *deduction function* $\mathcal{D}$ s.t. if ϕ is a finite set of closed messages, then $\mathcal{D}(\phi)$ is the set of closed messages that can be deduced starting from ϕ by applying instances of the rules in the system.

Example 1. We do not explicitly define an equality check among messages in the syntax. However, this can be implemented through the usage of the inference construct. E.g., consider rule $equal \doteq \frac{x \quad x}{Equal(x,x)}$. Then $[m = m']A$ (with the

$$\frac{m \quad m'}{(m,m')}(\vdash_{pair}) \qquad \frac{(m,m')}{m}(\vdash_{fst}) \qquad \frac{(m,m')}{m'}(\vdash_{snd})$$

$$\frac{F \quad x}{F(x)}(\vdash_{hash}) \qquad \frac{m \quad k}{mac(m,k)}(\vdash_{mac})$$

Fig. 1. An example inference system.

expected semantics) may be equivalently expressed as $[\langle m\ m'\rangle \vdash_{equal} y]A$ where y does not occur in A.

The operational semantics of a *tCryptoSPA* term is described by means of the *labeled transition system* (*lts*, for short) $\langle \mathcal{P}, Act, \{\xrightarrow{a}\}_{a\in Act}\rangle$, where $\{\xrightarrow{a}\}_{a\in Act}$ is the least relation between *tCryptoSPA* processes induced by the axioms and inference rules of Figure 2.

The expression $P \overset{a}{\Rightarrow} P'$ is a shorthand for $P(\xrightarrow{\tau})^* P_1 \xrightarrow{a} P_2(\xrightarrow{\tau})^* P', a \neq \tau$, where $(\xrightarrow{\tau})^*$ denotes a (possibly empty) sequence of transitions labeled τ. The expression $P \Rightarrow P'$ is a shorthand for $P(\xrightarrow{\tau})^* P'$. Let $\gamma = a_1, \ldots, a_n \in (Act\backslash\{\tau\})^*$ be a sequence of actions; then $P \overset{\gamma}{\Rightarrow} P'$ iff there exist $P_1, \ldots, P_{n-1} \in \mathcal{P}$ such that $P \overset{a_1}{\Rightarrow} P_1 \overset{a_2}{\Rightarrow}, \ldots, P_{n-1} \overset{a_n}{\Rightarrow} P'$. Let $\mathbf{0}' \doteq tick.\mathbf{0}'$.

For timed behavioural relations among *tCryptoSPA* processes, we will be mainly interested in *timed trace* inclusions.

Definition 1. *For any $P \in \mathcal{P}$ the set $T(P)$ of timed traces associated with P is defined as follows $T(P) = \{\gamma \in (Act\backslash\{\tau\})^* \mid \exists P'.P \overset{\gamma}{\Rightarrow} P'\}$. The timed trace pre-order, denoted by $\leq_{ttrace}$, is defined as follows: $P \leq_{ttrace} Q$* iff *$T(P) \subseteq T(Q)$. P and Q are* timed trace equivalent*, denoted by $P =_{ttrace} Q$, if $T(P) = T(Q)$.*

We define the concept of weak simulation as usual.

Definition 2. *We say that a relation R among processes is a weak simulation, if for every $(P, Q) \in \mathcal{R}$ we have:*

- *If $P \xrightarrow{a} P', a \neq \tau$, then there exists Q' s.t. $Q \overset{a}{\Longrightarrow} Q'$ and $(P', Q') \in \mathcal{R}$.*
- *If $P \xrightarrow{\tau} P'$ then there exists Q' s.t. $Q \Longrightarrow Q'$ and $(P', Q') \in \mathcal{R}$.*

Let $\prec$ the union of all weak simulations among processes. Then, we have $\prec \subseteq \leq_{ttrace}$.

3 tGNDC

A general schema for the definition of timed security properties, called *timed Generalized Non Deducibility on Compositions* (*tGNDC* for short) has been proposed in [6]: a system S is $tGNDC^{\alpha}_{\lhd}$ *iff* for every enemy X the composition of the system with X satisfies the timed specification $\alpha(S)$. Basically, *tGNDC* guarantees that the timed property α is satisfied, with respect to the $\lhd$ timed

$$(input)\frac{m \in \mathcal{M}}{c(x).P \xrightarrow{c(m)} P[m/x]} \qquad (output)\frac{}{\overline{c}m.P \xrightarrow{\overline{c}m} P} \qquad (internal)\frac{}{\tau.P \xrightarrow{\tau} P}$$

$$(tick)\frac{}{tick.P \xrightarrow{tick} P} \qquad (|)_1\frac{P_1 \xrightarrow{l} P_1'}{P_1 \,|\, P_2 \xrightarrow{l} P_1' \,|\, P_2} \qquad (|)_2\frac{P_1 \xrightarrow{c(x)} P_1' \quad P_2 \xrightarrow{\overline{c}m} P_2'}{P_1 \,|\, P_2 \xrightarrow{\tau} P_1' \,|\, P_2'}$$

$$(|)_3\frac{P_1 \xrightarrow{tick} P_1' \quad P_2 \xrightarrow{tick} P_2'}{P_1 \,|\, P_2 \xrightarrow{tick} P_1' \,|\, P_2'} \qquad (\backslash L)\frac{P \xrightarrow{c(m)} P' \quad c \notin L}{P\backslash L \xrightarrow{c(m)} P'\backslash L} \qquad (+_1)\frac{P_1 \xrightarrow{l} P_1'}{P_1 + P_2 \xrightarrow{l} P_1'}$$

$$(+_2)\frac{P_1 \xrightarrow{tick} P_1' \quad P_2 \xrightarrow{tick} P_2'}{P_1 + P_2 \xrightarrow{tick} P_1' + P_2'} \qquad (+_3)\frac{P_1 \xrightarrow{tick} P_1' \quad P_2 \not\xrightarrow{tick} \quad P_2 \not\xrightarrow{\tau}}{P_1 + P_2 \xrightarrow{tick} P_1'}$$

$$(Def)\frac{P[m_1/x_1, \ldots, m_n/x_n] \xrightarrow{a} P' \quad A(x_1, \ldots, x_n) \doteq P}{A(m_1, \ldots, m_n) \xrightarrow{a} P'}$$

$$(\mathcal{D})\frac{\langle m_1, \ldots, m_r\rangle \vdash_{rule} m \quad P_1[m/x] \xrightarrow{a} P_1'}{[\langle m_1, \ldots, m_r\rangle \vdash_{rule} x]P_1; P_2 \xrightarrow{a} P_1'}$$

$$(\mathcal{D})\frac{\nexists m \text{ s.t. } \langle m_1, \ldots, m_r\rangle \vdash_{rule} m \quad P_2 \xrightarrow{a} P_2'}{[\langle m_1, \ldots, m_r\rangle \vdash_{rule} x]P_1; P_2 \xrightarrow{a} P_2'}$$

Fig. 2. Structured Operational Semantics for tCryptoSPA (symmetric rules for $+_1, +_3, |_1, |_2$ and $\backslash L$ are omitted).

behavioural relation, even when the system is composed with any possible adversary X.

We give here the set of admissible hostile environments for our timed setting. For a certain enemy X, we call $ID(X)$ the set of closed messages that syntactically appears in X, all the messages initially known by X. Let ϕ_0 be the initial knowledge we would like to give to the enemy at the beginning of the computation. We require that all the messages in $ID(X)$ are deducible from ϕ_0. We consider as hostile processes only the ones belonging to the set $t\mathcal{E}_C^{\phi_0}$ [1]. They can communicate on a subset of public channels C and have an initial knowledge bound by ϕ_0:

$$t\mathcal{E}_C^{\phi_0} = \{X \in \mathcal{P} \mid sort(X) \subseteq C \text{ and } ID(X) \subseteq \mathcal{D}(\phi_0)\}$$

The property $tGNDC_{\lhd}^{\alpha}$ is defined as follows:

Definition 3. *S is $tGNDC_{\lhd}^{\alpha}$* iff *$\forall X \in t\mathcal{E}_C^{\phi_0} : (S \,|\, X)\backslash C \;\lhd\, \alpha(S)$ where $\lhd$ is a timed behavioural relation between processes and $\alpha : \mathcal{P} \to \mathcal{P}$ is a function between processes defining the property specification for S as the process $\alpha(S)$.*

[1] Actually, there is another constraint that imposes that the enemy must eventually let time pass. This is however not useful for safety properties we are going to study in this paper and so it has been omitted for the sake of simplicity.

We may define several security properties through the t$GNDC$ schema, e.g. see [6]. For instance timed secrecy expresses that a certain message m is not known by the intruder within a certain amount of time, say at least n units of time. A specification α_{tSec} dealing with timed secrecy could be the following:

$$\begin{aligned} pub(m) &= \overline{public}(m).\mathbf{0}' + tick.pub(m) \\ \alpha_{tSec} &= tick_1 \dots tick_n.(pub(m)) \end{aligned}$$

where we assume *public* the unique not restricted channel. α_{tSec} let n units of time pass and then behaves like *pub(m)*, i.e. it could either sends m over *public* or it could let time pass and possibly sends m.

Note that the GNDC theory is now a well established approach for security analysis and it was developed for non-deterministic, probabilistic, real time and cryptographic frameworks, e.g. see [1, 4–7]. Here we present an extension of the compositional analysis in a real-time setting within the GNDC theory.

3.1 Time-Dependent Stability and Compositional Results

A compositional principle gives sufficient conditions to conclude that the parallel composition of two (or more) processes satisfies a certain property, provided that the single processes by themselves satisfy the same property. Compositional reasoning is often useful. An interesting application field is indeed the analysis of systems with an arbitrary number of components.

Here, we give a new result about conditions for safe composition of digital stream protocols where time plays an essential role. In order to achieve this result, we should refine the concept of stability defined in [6] basically requires that the intruder knowledge does not increase when composing the intruder process with a process P. If so, we call P a stable process. In [6] it was also noticed that if we assume that the intruder knowledge does not increase when composing the intruder process with P (i.e. $P \,|\, X$) and with Q (i.e. $Q \,|\, X$) (using the same communication channels) then the intruder knowledge does not increase when composing the intruder itself with the process $P \,|\, Q$. Unfortunately, such a form of stability is not time-dependent, i.e. it takes into account the same knowledge during all the temporal execution of the processes at stake. This does not make it feasible to check properties based on a timed notion of secrecy and, consequently, to check protocols as μTESLA, whose security features exactly depend on a form of timed secrecy. We give now a refined notion of stability, called time-dependent stability, that allows us to cope with timed secrecy and so also with security properties of protocols that rely on it.

We let γ be a sequence of actions (possibly empty) ranging over $Act \backslash \{\tau\}$. Let $\#^{tick}(\gamma)$ be the number of occurrences of *tick* actions in the sequence γ.

Definition 4. *Let X_ϕ be the closed term in X belonging to the messages deducible from ϕ. We say that a process P is time-dependent stable w.r.t. the sequence $\{\phi_i\}_{i \geq 0}$ if, whenever $(P \,|\, X_{\phi_0}) \backslash C \stackrel{\gamma}{\Longrightarrow} (P' \,|\, X'_{\phi'}) \backslash C$ and $\#^{tick}(\gamma) = i$, then $\mathcal{D}(\phi') = \mathcal{D}(\phi_i)$.*

Basically, a process P is time-dependent stable if an enemy cannot increase significantly its knowledge when P runs in the space of a time slot. The following proposition holds:

Proposition 1. *Given a sequence* $\{\phi_i\}_{i\geq 0}$ *and a set of public channels* C*, assume* $P_r \in tGNDC^{\alpha_r(P_r)}_{\leq ttrace}$ *with* $1 \leq r \leq n$*. Assume also* P_r *t. d. stable w.r.t.* $\{\phi_i\}$*. It follows that* $(P_1 \,|\, P_2 \,|\, \ldots \,|\, P_n) \in tGNDC^{\alpha_1(P_1)\,|\,\alpha_2(P_2)\,|\ldots|\,\alpha_n(P_n)}_{\leq ttrace}$ *and* $(P_1 \,|\, P_2 \,|\, \ldots \,|\, P_n)$ *is t. d. stable w.r.t.* $\{\phi_i\}$*.*

Example 2. The process $P = tick.\bar{c}k.\mathbf{0}'$ enjoys the secrecy of k for one time unit. In the more complex process $Q = (c(x).[x = k]\bar{c}m) + tick.\mathbf{0}'$ the secrecy of k in the first time unit is crucial to get the secrecy of m. Indeed, either Q is willing to receive the key k only in the first time unit (if so, it releases m) or it starts to idle. We have that P and Q are t.d. stable w.r.t. $\phi_0 = \{\emptyset\}$, $\phi_i = \mathcal{D}(\{k\})$ for $i \geq 1$. Then, $P \,|\, Q$ is t.d. stable w.r.t. $\{\phi_i\}_{i\geq 0}$ (by Proposition 1) and so m will never belong to the knowledge of the intruder (whose initial knowledge is $\emptyset$).

4 The μTESLA Protocol

In [13], Perrig *et al.* presented μTESLA ("micro" Timed Efficient Stream Loss-tolerant Authentication), a protocol to provide authenticated broadcast in wireless sensor networks environments. [13] considers a scenario where sensors communicate with a base-station connected to the external world. The base station may broadcast to all nodes messages for routing updates, reprogramming, reset requests. The protocol is an extension of the TESLA stream authentication protocol developed in [12] and it was intentionally developed for providing authenticated broadcast for the limited computing environments that are encountered in sensor networks.

In the original TESLA schema, a single sender broadcasts a continuous stream of packets. Receivers may use information in later packets to authenticate earlier packets. Each packet contains a message authentication code (MAC), i.e. a value computed by applying a public algorithm and a secret encryption key to the packet itself. Given a message m and an encryption key k, we call *mac(m, k)* the message authentication code of m. The algorithm is known by all the receivers, while the encryption keys are disclosed by the sender after a certain amount of time. When a receiver receives a key K_i it can use it to compute the MAC from the related packet P_i and compare the computed MAC with that previously received. If the two MACs match, the receiver can consider the packet P_i authentic. To avoid the event that an intruder could use a disclosed key K_i to fake the packet P_i a time synchronization protocol between the sender and the receivers is needed. Then, each receiver will not accept the packet P_i if the sender might have already sent the key K_i.

Bootstrapping authentication of the whole scheme is achieved in TESLA by signing the first packet with a regular digital signature scheme. Nevertheless, computation, communication and storage overhead make the use of asymmetric cryptography unfeasible for the net of sensors under investigation. Thus,

μTESLA has been proposed as an optimized extension for sensor networks. It just makes use of MACs. The base-station randomly generates the last MAC key to be used, K_{last}, and derives a key chain by repeatedly applying a publicly known one-way function F to that key, such that $K_i = F(K_{i+1})$. Given the non-reversibility property (at least with high probability) of function F, the disclosure of key K_i should not lead to any knowledge of K_{i+1} and subsequent keys.

Receivers' requirements for correctly joining and executing the protocol are: i) they are time synchronized with the base station; ii) they know the disclosure schedule of the MAC keys; iii) they know at least one authenticated key of the key chain, serving as a commitment to the entire chain. A protocol providing time synchronization and one authenticated key has been proposed in [13]. Basically, the base-station shares with each sensor a symmetric secret key K_{SM} and establishes a secure channel over which the exchange of a commitment to the key chain, K_0, and a set of temporal parameters, set_t, takes place[2]. More formally, the initial step of μTESLA is the following:

$$\text{Packet } P_0 \quad c_0\ S \rightarrow \{R_n\} : K_0, set_t, mac(K_0, set_t, K_{SM})$$

where $c_0 \in \{c_i\}_{i \in \mathbb{N}}$, i.e. the set of communication channels, S is the identifier of the sender[3] (i.e. the base station) and $\{R_n\}$ is the set of receivers (i.e. the sensors).

μTESLA is parameterized by the schedule time at which MAC keys are disclosed. For the description of further steps in the protocol we consider a basic formalization, Fig. 3, where we suppose that the sender discloses a MAC key with a delay $\delta = 1$, assumed to fall in the interval after that key has been used to compute the MAC. Further, we suppose the sender sends one packet per time interval. Basically, in each time slot a packet and a key packet will be sent, Fig. 3. First of all, each receiver should check the integrity of the received key, say K_i, by verifying it w.r.t. an authenticated commitment (e.g. by checking $K_0 = F^i(K_i)$), then the verified key will be used to verify the integrity of the packet received in the previous time slot.

$$\text{Packet } P_i \quad c_i\ S \rightarrow \{R_n\} : m_i, mac(m_i, K_i)\ i \geq 1$$

Packet P_i consists of a meaningful payload m_i plus the message authentication code computed on m_i with key K_i. We assume that K_{SM} cannot be deduced from the sets $\{m_i\}, \{K_i\}$.

[2] There are as many symmetric keys as the number of sensors and the communication over channel c_0 is supposed to be a point to point communication. Nevertheless, to simplify our formalization, we assume a unique key and a unique communication. This means to implicitly assume that possible adversaries are not in the set of receivers.

[3] To assure freshness when executing multiple runs of the same sender, one can simply insert nonces in the message authentication code of packet P_0.

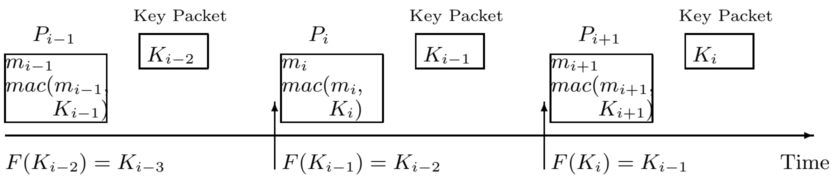


Fig. 3. A μTESLA instantiation.

Upon receiving the packet, the sensor stores the packet until its MAC can be verified, i.e. until the sender broadcasts packet disclosing K_i:

$$\text{Key-Packet } KP_i \quad c_{i+1}\ S \to \{R_n\} : K_i$$

The integrity of key K_i can be checked by verifying $K_0 = F^i(K_i)$ (or, equivalently, $K_{i-1} = F(K_i)$). Packets may be lost in transit from the base station to the sensors. In particular μTESLA is tolerant to packet loss in the sense that receivers may still be able to authenticate all the received packets P_i even when the corresponding keys' disclosure packets are lost. Suppose K_j is lost, then a receiver is not able to verify MAC packet P_j. The following key the receiver recovers, let it be K_{j+1}, can be verified w.r.t. a previous authenticated key (e.g. $K_0 = F^{j+1}(K_{j+1})$) and is used to derive K_j, i.e. $K_j = F(K_{j+1})$.

4.1 The *tCryptoSPA* Specifications of the μTESLA Protocol

We present the *tCryptoSPA* specifications of the basic μTESLA instantiation in Fig. 3. The fundamental requirement of a time synchronization between a base-station and each sensor in μTESLA is naturally captured in *tCryptoSPA* by its time modeling action *tick*, upon which sender and receivers' processes may synchronize (this allows us to avoid the explicit presence of set_t in packet P_0).

We consider a sender machine with ample resources. It can be parallelized or split into n senders, each of them possibly sending different streams, $\{m_i^j\}_{i \geq 1, 1 \leq j \leq n}$. We first present the generic sender process S^j, parameterized by a sequence of MAC keys (tied together by means of a key chain)[4]. We assume the symmetric key K_{SM}, the keys belonging to the key chain and the streams of packets to be different for each process $S^j, 1 \leq j \leq n$[5].

$$\begin{array}{ll}
S_0^j(K_{SM}^j, K_0^j, K_1^j, \ldots) \doteq & \\
[K_0^j \quad K_{SM}^j \vdash_{mac} y] & \text{Compute MAC} \\
[K_0^j \quad y \vdash_{pair} P_0] & \text{Create packet } P_0 \\
B_0^j(P_0) & \text{Start to broadcast } P_0
\end{array}$$

[4] Actually, we consider constants with an arbitrary number of parameters. We could avoid this by considering, for modeling purposes, a special function fun, not available to possible adversaries, that may be used to represent the keys as a sequence.

[5] We remind the reader that the whole formalization we are going to give is based on choices of the authors since some details are not explicitly given in [13]. In particular, the mechanism through which a receiver possibly identifies each sender process (and consequently each stream) is not defined in [13], since the original construction is described with a single sender.

$$\begin{array}{ll} S_1^j(K_0^j, K_1^j, \ldots) \doteq & \\ [m_1^j \quad K_1^j \vdash_{mac} x] & \text{Compute MAC} \\ [m_1^j \quad x \vdash_{pair} P_1] & \text{Create packet } P_1 \\ B_1^j(P_1) & \text{Start to broadcast } P_1 \end{array}$$

$$\begin{array}{ll} S_i^j(K_{i-1}^j, K_i^j, \ldots) \doteq & \\ [m_i^j \quad K_i^j \vdash_{mac} x] & \text{Compute MAC} \\ [m_i^j \quad x \vdash_{pair} P_i] & \text{Create packet } P_i \\ B_i^j(P_i, K_{i-1}^j) & \text{Start to broadcast } P_i \text{ and disclose key } K_{i-1} \end{array}$$

$$B_i^j(P_i) \doteq \overline{c_i}P_i.B_i^j(P_i) + tick.S_{i+1}^j(K_i^j, \ldots) \quad i = 0, 1$$
$$B_i^j(P_i, K_{i-1}^j) \doteq \overline{c_i}P_i.\overline{c_i}K_{i-1}^j.B_i^j(P_i, K_{i-1}^j) + tick.S_{i+1}^j(K_i^j, \ldots) \quad i \geq 2$$

Construct $B_i^j(\ldots)$ is responsible for potentially sending packets (and keys) an unbounded number of times, in order to simulate a one-to-all sending typical of broadcast sessions. Sender S^j remains in the same state repeatedly sending messages unless the non-deterministic choice is resolved by choosing the derivative of the second summand in B_i^j; this causes a time unit to pass (a *tick* action is performed). The construction models the behaviour of a wireless antenna making signals available only in a particular time interval. The presence of a non-deterministic choice in the construct makes it possible the passage to the following time interval without performing any (eventually zero) communication. This may implicitly model the unreliability of the wireless transmission and the occurrence of packet loss.

Among the receivers' set, each process behaves in the same way. The generic receiver process at step i is parameterized by a commitment to the key chain (let it be K_0^j) and by the packets it should still authenticate. We assume the receiver's set is divided into subgroups, each of them sharing a particular K_{SM} with one sender process. Sender S^j and receivers belonging to subgroup number j share K_{SM}^j. K_{SM}^j may denote a particular service each element in subgroup j is devoted to. Let us consider pay per view-based applications: among the receivers' set, the subgroup knowing K_{SM}^j may consist of all the paying spectators for movie number j. For environments closer to those depicted for μ-TESLA, let us consider a scenario in which sensors are used to periodically transmit readings regarding heating and air conditioning control in a building (and consequently receive broadcasted messages for routing updates or reprogramming): sensors in subgroup j may be all the sensors devoted to carry out the service for room number j. (S^j being the base station responsible for room number j.).

Below, we refer to $R_i^{j,q}$ to indicate the q-th receiver process belonging to subgroup j and acting at step i.

$$\begin{array}{ll} R_0^{j,q}(null) \doteq & \\ c_0(x). & \text{Receive first packet} \\ [x \vdash_{fst} x_{K_0}] & \text{Extract commitment to the key chain} \\ [x_{K_0} \quad K_{SM}^j \vdash_{mac} z] & \text{Compute MAC} \end{array}$$

$[x \vdash_{snd} x_{mac}]$	Extract MAC
$[z = x_{mac}]$	Verify MAC: if verified:
$tick.R_1^{j,q}(x_{K_0});$	Allow a time unit to pass and go to next state
$R_0^{j,q}(null)$	Wait for key

Upon receiving a value x on channel c_0, the receiver verifies the correctness of the commitment to the key chain, x_{K_0}: it computes $mac(x_{K_0}, K_{SM}^j)$ and compares it with the message authentication code in the received packet. If the two MACs match, a time unit passes and the receiver goes to the next state, otherwise the receiver remains in the same state waiting for the right key K_{SM}^j. Throughout the formalization, *null* means an empty field.

$R_1^{j,q}(x_{K_0}) \doteq$	
$(c_1(y).$	Receive packet
$tick.R_2^{j,q}(y, x_{K_0})$	Allow a time unit to pass and go to next state
$) + tick.R_2^{j,q}(null, x_{K_0})$	Go to next state after a time unit

$R_1^{j,q}$ is willing to accept any arbitrary packet, because it cannot perform any verification yet. If nothing is received before the end of a time unit, transition takes place to next state $R_2^{j,q}$.

$R_i^{j,q}(p_{i-1}, x_{K_0}) \doteq$	
$c_i(p_i).R_i'^{j,q}(p_i, p_{i-1}, x_{K_0})$	Receive i-th packet; go to intermediary state $R_i'^{j,q}$
$+tick.R_{i+1}^{j,q}(null, x_{K_0})$	Go to next state after a time unit

$R_i^{j,q}$ is willing to accept packet P_i and travels to an intermediary state $R_i'^{j,q}$. If nothing is received before the end of a time unit, transition takes place to the next state.

$R_i'^{j,q}(p_i, p_{i-1}, x_{K_0}) \doteq$	
$c_i(x_{K_{i-1}}).$	Receive key packet
$[x_{K_0} = F^{i-1}(x_{K_{i-1}})]$	Verify the key w.r.t. the commitment
$[p_{i-1} \vdash_{fst} y_{pay}]$	Extract payload
$([y_{pay} \quad x_{K_{i-1}} \vdash_{mac} z]$	If $x_{K_{i-1}} = K_{i-1}^j$ then: Compute MAC
$[p_{i-1} \vdash_{snd} y_{mac}]$	Extract MAC
$[z = y_{mac}]$	Verify MAC
$\overline{appy} y_{pay}.$	Send m_1^j to application level
$tick.R_{i+1}^{j,q}(p_i, x_{K_0})$	Allow a time unit to pass and go to next state
$); R_i'^{j,q}(p_i, p_{i-1}, x_{K_0})$	Wait for key

In intermediary state $R_i'^{j,q}$ receives a key packet and verifies the correctness of the key w.r.t. the authenticated commitment $x_{K_0} = K_0^j$. Given the collision-free property of one-way functions, if the verification does not succeed it means $x_{K_{i-1}} \neq K_{i-1}^j$ and $R_i'^{j,q}$ simply stays in the same state waiting for the right subgroup key. If the verification succeeds, the correctness of P_{i-1} is verified by checking that the enclosed MAC is authentic. The successful outcome is here modeled by a scenario where the receiver sends the payload of the accepted packet over channel *app*[6].

Suppose packet P_{i-1} was correctly received, suppose also packet disclosing K^j_{i-1} is lost. At step i the receiver still cannot authenticate packet P_{i-1}. The key chain mechanism of the original protocol takes into account such a possibility: in interval $i+1$ the base station broadcasts key K^j_i, which the receiver authenticates by verifying $K^j_0 = F^i(K^j_i)$. The receiver can authenticate P_i and derives $K^j_{i-1} = F(K^j_i)$, so it can also authenticate P_{i-1}. Actually, our formalization does not take into account recovering lost keys. For the sake of simplicity, we prefer to suppose that the key packet related to subgroup j is received (state $R'^{j,q}_i$).

We report below the formalization at step i, with $i \geq 2$, when a packet was not received at step $i-1$.

$$\begin{array}{ll} R^{j,q}_i(null, x_{K_0}) \doteq & \\ c_i(p_i).tick.R^{j,q}_{i+1}(p_i, x_{K_0}) & \text{Receive i-th packet; go to next state} \\ +tick.R^{j,q}_{i+1}(null, x_{K_0}) & \text{Go to next state after a time unit} \end{array}$$

5 An Analysis of the μTESLA Protocol: Timed Integrity

We focus our attention on the so called timed integrity, belonging to a new class of properties defined in [6]. A stream signature protocol guarantees timed integrity on a set of messages $\{m_i\}$ if, whenever the generic receiver accepts an item in a time interval i, let us say item x, then $x = m_{i-\delta}$, $i - \delta$ being the time interval in which x has been received. ($\delta = 1$ in our formalization of μTESLA.) Timed integrity property may be efficiently verified by means of Proposition 1 in Subsection 3.1. We consider μTESLA as a case study for proving its correctness in terms of messages m_i timed integrity. We refer to the instantiation in Fig. 3 and its *tCryptoSPA* specifications of Subsection 4.1. Assume that a receiver signals the acceptance of a payload as a legitimate one, by issuing it on a special channel *app*.

Let $P^q \doteq S^j_0 \mid R^{j,q}_0$ be the system consisting of a single sender and q-th receiver in subgroup j, sharing K^j_{SM}. Let function $\alpha_{tInt}(P^q)$ be $tSpec_0$ where

$$\begin{array}{l} tSpec_0 \doteq tick.tSpec_1 \\ tSpec_1 \doteq tick.tSpec_2 \\ tSpec_i \doteq tick.tSpec_{i+1} + \overline{app}(m^j_{i-1}).tick.tSpec_{i+1} \; i \geq 2 \end{array}$$

$\alpha_{tInt}(P^q)$ may denote the correct external behaviour of P^q. In the first two steps it simply let time pass, while in further steps it may either let time pass (denoting packet loss) or let a verified payload to be sent on the special channel *app* and then let time pass. The set of all messages sent on channel *app* is the

[6] We omitted to insert an idling behavior when a deduction construct fails to be executed and in our formalization the system simply stops without letting time pass. This is not realistic, but it has no consequences since we use trace semantics for the analysis and makes it simpler.

set of all the possible ordered substreams of $\{m_i^j\}_{i\geq 1}$. Let function $\alpha_{tInt}^j(P^j) \doteq \Pi_{1\leq q\leq n_j}\alpha_{tInt}(P^q)$, n_j being the cardinality of the receivers in subgroup j.

Definition 5. *The system $P^j \doteq S_0^j \,|\, R_0^{j,1} \,|\, R_0^{j,2} \,|\, \ldots \,|\, R_0^{j,n_j}$, consisting of a sender of streamed data $\{m_i^j\}$ and the receivers in subgroup j enjoys the timed integrity property whenever $P^j \in tGNDC_{\leq ttrace}^{\alpha_{tInt}^j(P^j)}$.*

Basically, it means that each receiver accepts exactly the messages belonging to $\{m_i^j\}$ in the correct order and within the time interval following the one in which the sender actually sent the messages, even in presence of an intruder (unless packets P_i are lost). The key point is that the intruder will never acquire the shared key K_{SM}^j to establish a secure channel over which the commitment to the key chain is exchanged[7].

We first consider system P^q. We may prove that S_0^j and $R_0^{j,q}$ (Subsection 4.1) are t.d. stable w.r.t. the sequence $\{\phi_i\} = \phi_0, \phi_1, \phi_2, \ldots$ defined as follows:

$$\begin{aligned}
&\phi_0 = \{K_0^j, mac(K_0^j, K_{SM}^j) \mid 1 \leq j \leq n\}\\
&\phi_1 = \phi_0 \cup \{m_1^j, mac(m_1^j, K_1^j) \mid 1 \leq j \leq n\}\\
&\phi_2 = \phi_1 \cup \{m_2^j, mac(m_2^j, K_2^j), K_1^j \mid 1 \leq j \leq n\}\\
&\ldots\\
&\phi_i = \phi_{i-1} \cup \{m_i^j, mac(m_i^j, K_i^j), K_{i-1}^j \mid 1 \leq j \leq n\}\\
&\ldots
\end{aligned}$$

where n is the number of senders. ϕ_i is equal to ϕ_{i-1} plus the set of all the messages an intruder would be able to add to its knowledge by eavesdropping on a run of the protocol during the whole time interval i (of course including those messages coming from all the other senders processes). Actually, the intruder will have more powerful means to act since the beginning of each time interval.

We may prove that S_0^j enjoys $tGNDC_{\leq ttrace}^{\mathbf{0}'}$ and $R_0^{j,q}$ enjoys $tGNDC_{\leq ttrace}^{\alpha_{tInt}(P^q)}$, that is to say for all $X \in t\mathcal{E}_C^{\phi_0}$ we have $(S_0^j \,|\, X)\backslash C \leq_{ttrace} \mathbf{0}'$ and $(R_0^{j,q} \,|\, X)\backslash C \leq_{ttrace} \alpha_{tInt}(P^q)$. This may be done by finding a suitable weak simulation relation between $(S_0^j \,|\, X_{\phi_0}) \setminus C$ and $\mathbf{0}'$ and between $(R_0^{j,q} \,|\, X_{\phi_0}) \setminus C$ and $tSpec_0$, respectively. The set C of channels over which an intruder is able to communicate is $C = \{c_i \mid i \geq 0\}$. The weak simulation relation dealing with the sender specifications is the following:

$$\begin{aligned}
\mathcal{R}_{\mathcal{S}} = &(((S_i^j(\ldots) \,|\, X_{\phi_i})\backslash C, \mathbf{0}') \mid \forall i, X_{\phi_i} \in t\mathcal{E}_C^{\phi_i})\\
&\cup(((B_i^j(\ldots) \,|\, X_{\phi_i})\backslash C, \mathbf{0}') \mid \forall i, X_{\phi_i} \in t\mathcal{E}_C^{\phi_i})\\
&\cup(((\overline{c_i}K_{i-1}^j.B_i^j(\ldots) \,|\, X_{\phi_i})\backslash C, \mathbf{0}') \mid i > 1, X_{\phi_i} \in t\mathcal{E}_C^{\phi_i})
\end{aligned}$$

[7] We remind the reader that $K_{SM}^m \neq K_{SM}^n$ if $m \neq n$ and $K_i^m \neq K_l^n$ if $m \neq n$ or $i \neq l$.

The weak simulation relation we consider for dealing with the receiver specifications is the following (superscript q is omitted for simplicity):

$$\begin{aligned}\mathcal{R} = {} & (((R_0^j(null) \,|\, X_{\phi_0})\backslash C, tSpec_0) \mid X_{\phi_0} \in t\mathcal{E}_C^{\phi_0}) \\ & \cup((tick.(R_1^j(K_0^j) \,|\, X_{\phi_0})\backslash C, tSpec_0) \mid X_{\phi_0} \in t\mathcal{E}_C^{\phi_0}) \\ & \cup(((R_1^j(K_0^j) \,|\, X_{\phi_1})\backslash C, tSpec_1) \mid X_{\phi_1} \in t\mathcal{E}_C^{\phi_1}) \\ & \cup(((R_i^j(null, K_0^j) \,|\, X_{\phi_i})\backslash C, tSpec_i) \mid i \geq 2, X_{\phi_i} \in t\mathcal{E}_C^{\phi_i}) \\ & \cup((tick.(R_i^j(p_{i-1}, K_0^j) \,|\, X_{\phi_{i-1}})\backslash C, tSpec_{i-1}) \mid i \geq 2, X_{\phi_{i-1}} \in t\mathcal{E}_C^{\phi_{i-1}}) \\ & \cup(((R_i^j(p_{i-1}, K_0^j) \,|\, X_{\phi_i})\backslash C, tSpec_i) \mid i \geq 2, X_{\phi_i} \in t\mathcal{E}_C^{\phi_i}) \\ & \cup(((R_i^{j'}(p_i*, p_{i-1}, K_0^j) \,|\, X_{\phi_i})\backslash C, tSpec_i) \mid i \geq 2, X_{\phi_i} \in t\mathcal{E}_C^{\phi_i}) \\ & \cup(((R_i^j(x_{i-1}, K_0^j) \,|\, X_{\phi_i})\backslash C, tSpec_i) \mid fst(x_{i-1}) \neq m_{i-1}^j, i \geq 2, X_{\phi_i} \in t\mathcal{E}_C^{\phi_i}) \\ & \cup((tick.(R_i^j(x_{i-1}, K_0^j) \,|\, X_{\phi_{i-1}})\backslash C, tSpec_{i-1}) \mid fst(x_{i-1}) \neq m_{i-1}^j, i \geq 2, \\ & X_{\phi_{i-1}} \in t\mathcal{E}_C^{\phi_{i-1}}) \\ & \cup((tick.(R_i^j(p_{i-1}*, K_0^j) \,|\, X_{\phi_{i-1}})\backslash C, tick.tSpec_i) \mid i \geq 2, X_{\phi_{i-1}} \in t\mathcal{E}_C^{\phi_{i-1}})\end{aligned}$$

where $p_1, p_{i-1}, p_i*, p_{i-1}*$ and x_{i-1} are not empty fields. $p_i*, p_{i-1}*$ are shortcuts to denote either authentic packets sent by the sender or others. We omitted to explicitly put in $\mathcal{R}_{\mathcal{S}}$ and $\mathcal{R}$ the pairs in which the first process performs deduction constructs.

Lemma 1. *S_0^j and $R_0^{j,q}$ are t. d. stable w.r.t. $\{\phi_i\}$.*

Lemma 2. *$S_0^j \in tGNDC_{\leq ttrace}^{\mathbf{0}'}$ and $R_0^{j,q} \in tGNDC_{\leq ttrace}^{\alpha_{tInt}(P^q)}$*

The following proposition follows by Lemma 1,2 and by Proposition 1 where $r = 1, 2$, $P_1 = S_0^j$, $P_2 = R_0^{j,q}$.

Proposition 2. *$P^q \in tGNDC_{\leq ttrace}^{\alpha_{tInt}(P^q)}$*[8]*.*

The correctness of the multiple receivers version (considering all the receivers belonging to subgroup j), can be also proved using results of Lemma 1,2 and Proposition 1 where index r is not fixed *a priori* and $P_1 = S_0^j$ and $P_r = R_0^{j,q}$ with $1 \leq q \leq n_j$.

Proposition 3. *System P^j (in Definition 5) $\in tGNDC_{\leq ttrace}^{\alpha_{tInt}^j(P^j)}$.*

We get into the issue of considering a multiple senders/receivers environment. Let us consider $\Gamma = \Pi_{1 \leq j \leq n} P^j$ and $\alpha_{tInt}(\Gamma) = \Pi_{1 \leq j \leq n} \alpha_{tInt}^j(P^j)$, where n is the cardinality of the senders processes.

Proposition 4. *System $\Gamma \in tGNDC_{\leq ttrace}^{\alpha_{tInt}(\Gamma)}$.*

The result follows by application of Propositions 3 and 1.

Note that in order to have timed integrity on the messages m_i, μTESLA must ensure timed secrecy on the keys K_i. Indeed, we could also check explicitly timed secrecy on the keys with the same machinery.

[8] Note that $\mathbf{0}' \,|\, \alpha_{tInt}(P^q) \leq_{ttrace} \alpha_{tInt}(P^q)$.

6 Conclusions

In this paper we presented some preliminary steps towards a framework suitable for the security analysis of time-dependent wireless protocols. In particular, we developed a compositional approach for reasoning about security properties that rely on time constraints. This allowed us to check a relevant protocol, i.e. μTESLA. As a future work, we plan to deal with security properties in a mobile framework and offering some tool support for our compositional analysis.

Related work in security protocol verification in a timed setting may be found in [15], where tock-CSP is presented. The main differences are a different treatment of time operators and cryptography modeling. Moreover, no compositional proof rule has been provided. However, the verification proposed in [15] is automated through the use of PVS ([16]) while ours is completely manual (as of now).

Acknowledgments. We would like to thank the anonymous referees for their helpful comments.

References

1. A. Aldini, M. Bravetti, and R. Gorrieri. A Process-algebraic Approach for the Analysis of Probabilistic Non-interference. *Journal of Computer Security*, 2003.
2. P. Broadfoot and G. Lowe. Analysing a Stream Authentication Protocol using Model Checking. In *Proc. of ESORICS'02, LNCS 2502, 146-161*, 2002.
3. R. Focardi, R. Gorrieri, and F. Martinelli. Non Interference for the Analysis of Cryptographic Protocols. In *Proc. of ICALP'00, LNCS 1853, 354-372*, 2000.
4. R. Focardi, R. Gorrieri, and F. Martinelli. Real-Time Information Flow Analysis. *IEEE JSAC*, 21(1), 2003.
5. R. Focardi and F. Martinelli. A uniform approach for the definition of security properties. In *Proc. of FM'99, LNCS 1708, 794-813*, 1999.
6. R. Gorrieri, E.Locatelli, and F.Martinelli. A Simple Language for Real-time Cryptographic Protocol Analysis. In *Proc. of ESOP'03, LNCS 2618, 114-128*, 2003.
7. R. Gorrieri and F. Martinelli. Process Algebraic Frameworks for the Specification and Analysis of Cryptographic Protocols. In *Proc. of MFCS, LNCS 2747*, 2003.
8. R. Gorrieri, F. Martinelli, M.Petrocchi, and A.Vaccarelli. Compositional Verification of Integrity for Digital Stream Signature Protocols. In *Proc. of IEEE ACSD'03, 142-149*, 2003.
9. M. Hennessy and T. Regan. A Temporal Process Algebra. *I&C*, 117:222–239, 1995.
10. Y. W. Law, S. Dulman, S. Etalle, and P. Havinga. Assessing Security in Energy-efficient Sensor Networks. In *Proc. of Small Systems Security Workshop'03*, 2003.
11. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
12. A. Perrig, R. Canetti, D. X. Song, and D. Tygar. Efficient and Secure Source Authentication for Multicast. In *Proc. of NDSS'01.* The Internet Society, 2001.
13. A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. Culler. SPINS: Security Protocols for Sensor Networks. *Wireless Networks Journal*, 8:521–534, 2002.
14. K. Romer. Time Synchronization in Ad Hoc Networks. In *Proc. of ACM MobiHoc'01*, pages 173–182, 2001.
15. S. Schneider. Analysing Time-Dependent Security Properties in CSP using PVS. In *Proc. of ESORICS'00, LNCS 1895*, 2000.
16. N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial.* Tutorial Notes, *FME '93: Industrial-Strength Formal Methods*, pages 357–406, April 1993.

Inductive Proof Outlines for Monitors in Java[*]

Erika Ábrahám[1], Frank S. de Boer[2],
Willem-Paul de Roever[1], and Martin Steffen[1]

[1] Christian-Albrechts-University Kiel, Germany
`{eab,wpr,ms}@informatik.uni-kiel.de`
[2] CWI Amsterdam, The Netherlands
`F.S.de.Boer@cwi.nl`

Abstract. The research concerning *Java*'s semantics and proof theory has mainly focussed on various aspects of sequential sub-languages. *Java*, however, integrates features of a class-based object-oriented language with the notion of multi-threading, where multiple threads can concurrently execute and exchange information via shared instance variables. Furthermore, each object can act as a monitor to assure mutual exclusion or to coordinate between threads.
In this paper we present a sound and relatively complete assertional proof system for *Java*'s monitor concept, which generates verification conditions for a concurrent sublanguage $Java_{MT}$ of *Java*. This work extends previous results by incorporating *Java*'s monitor methods.

Keywords: OO, Java, multithreading, monitors, deductive verification, proof-outlines

1 Introduction

From its inception, *Java* [12] has attracted interest from the formal methods community: The widespread use of *Java* across platforms made the need for formal studies and verification support more urgent, the grown awareness and advances of formal methods for real-life languages made it more acceptable, and last but not least the array of non-trivial language features made it challenging.

Nevertheless, research concerning *Java*'s proof theory concentrated mainly on various aspects of *sequential* sub-languages (see e.g. [14, 23, 20]). In [5], we presented a sound and complete proof method for multithreaded *Java*, where threads can concurrently execute and exchange information via shared instance variables. In this paper we extend our results to deal with *Java*'s *monitor synchronization* mechanism: each object can act as a monitor to assure mutual exclusion or to coordinate between threads.

To support a clean interface between internal and external object behavior, we exclude qualified references to instance variables. As a consequence, shared-variable concurrency is caused by simultaneous execution within a single object,

[*] Part of this work has been financially supported by IST project Omega (IST-2001-33522) and NWO/DFG project Mobi-J (RO 1122/9-1, RO 1122/9-2).

only, but not across object boundaries. In order to capture program behavior in a modular way, the assertional logic and the proof system are formulated in two levels, a local and a global one. The local assertion language describes the internal object behavior. The global behavior, including the communication topology of the objects, is expressed in the global language. As in the Object Constraint Language (OCL) [24], properties of object-structures are described in terms of a navigation or dereferencing operator.

The assertional proof system for safety properties is formulated in terms of *proof outlines* [19], i.e., of programs augmented by auxiliary variables and annotated with Hoare-style assertions [11, 13]. Invariance of the asserted program properties is guaranteed by the verification conditions of the proof system. The execution of a single method body in isolation is captured by standard *local correctness* conditions, using the local assertion language. Interference between concurrent method executions is covered by the *interference freedom test* [19, 16], formulated also in the local language. It has especially to accommodate for reentrant code and the specific synchronization mechanism. Possibly affecting more than one instance, communication and object creation are treated in the *cooperation test*, using the global language. The theory presented here forms the theoretical foundation for a verification tool (*Verger*) which takes asserted *Java* programs as input and generates verification conditions for the *PVS* theorem prover as output; the use of the tool on a number of examples is reported in [4].

Overview. The paper is organized as follows. Section 2 defines syntax and semantics of the programming language. After introducing the assertional logic in Section 3, the main Section 4 presents the proof system. Section 5 discusses related and future work.

2 The Programming Language *Java*$_{MT}$

Similar to *Java*, the language *Java*$_{MT}$ is strongly typed; besides class types c, it supports booleans and integers as primitive types, and products and lists as composite types. Each corresponding value domain is equipped with a standard set of operators with typical element f; we use $\alpha, \beta, \ldots$ as typical object identities.

2.1 Syntax

The *Java*$_{MT}$ syntax is summarized in Table 1. We notationally distinguish between *instance* variables $x \in IVar$, and stack-allocated *local* or *temporary* variables $u \in TVar$ of methods; we use y to denote variables from $Var = IVar \dot{\cup} TVar$. Programs are collections of classes containing method declarations, where we use c and m for class names resp. method names, and $body_{m,c}$ to refer to the body of method m in class c. *Instances* of the classes, i.e., *objects,* are dynamically created, and communicate via *method invocation.* To increase readability of the proof outlines, we deviate from standard *Java* syntax, in that method invocation is syntactically split into a sending and a receiving statement.

Each class contains the predefined methods start, wait, notify, and notifyAll, and furthermore a user-defined run-method. The entry point of the program

Table 1. $Java_{MT}$ abstract syntax.

$$\begin{aligned}
exp &::= x \mid u \mid \mathsf{this} \mid \mathsf{nil} \mid \mathsf{f}(exp, \ldots, exp)\\
exp_{ret} &::= \epsilon \mid exp\\
u_{ret} &::= \epsilon \mid u\\
stm &::= \epsilon \mid x := exp \mid u := exp \mid u := \mathsf{new}^{c} \mid exp.m(exp, \ldots, exp); \mathsf{receive}\, u_{ret}\\
&\quad\ \mid\ exp.\mathsf{start}() \mid stm; stm \mid \mathsf{if}\ exp\ \mathsf{then}\ stm\ \mathsf{else}\ stm\ \mathsf{fi} \mid \mathsf{while}\ exp\ \mathsf{do}\ stm\ \mathsf{od} \ldots\\
modif &::= \mathsf{nsync} \mid \mathsf{sync}\\
meth &::= modif\, m(u, \ldots, u)\{\ stm; \mathsf{return}\ exp_{ret}\}\\
meth_{\mathsf{run}} &::= modif\, \mathsf{run}()\{\ stm; \mathsf{return}\ \}\\
meth_{\mathsf{start}} &::= \mathsf{nsync\ start}()\{\ \mathsf{this.run}(); \mathsf{receive}; \mathsf{return}\ \}\\
meth_{\mathsf{wait}} &::= \mathsf{nsync\ wait}()\{\ ?\mathsf{signal}; \mathsf{return}_{getlock}\ \}\\
meth_{\mathsf{notify}} &::= \mathsf{nsync\ notify}()\{\ !\mathsf{signal}\ ; \mathsf{return}\ \}\\
meth_{\mathsf{notifyAll}} &::= \mathsf{nsync\ notifyAll}()\{\ !\mathsf{signal\_all}; \mathsf{return}\ \}\\
meth_{\mathsf{main}} &::= \mathsf{nsync\ main}()\{\ stm; \mathsf{return}\ \}\\
class &::= c\{meth \ldots meth\ meth_{\mathsf{run}}\ meth_{\mathsf{start}}\ meth_{\mathsf{wait}}\ meth_{\mathsf{notify}}\ meth_{\mathsf{notifyAll}}\}\\
class_{\mathsf{main}} &::= c\{meth \ldots meth\ meth_{\mathsf{run}}\ meth_{\mathsf{start}}\ meth_{\mathsf{wait}}\ meth_{\mathsf{notify}}\ meth_{\mathsf{notifyAll}}\ meth_{\mathsf{main}}\}\\
prog &::= \langle class \ldots class\ class_{\mathsf{main}} \rangle
\end{aligned}$$

is given by the main-method of the program's main class. Invocation of the start-method, which can be done successfully only once, spawns a new thread of execution while the initiating thread continues its own execution.

As a mechanism of concurrency control, methods can be declared as *synchronized.* Each object has a *lock* which can be owned by at most one thread. Synchronized methods of an object can be invoked only by a thread that owns the lock of that object. Without the lock, a thread has to wait until the lock becomes free. The owner of an object's lock can recursively invoke several synchronized methods of that object, which corresponds to the notion of reentrant monitors. The monitor methods wait, notify, and notifyAll facilitate efficient thread coordination at the object boundary. Their definitions use the auxiliary statements !signal, !signal_all, ?signal, and $\mathsf{return}_{getlock}$. A thread owning the lock of an object can block itself and free the lock by invoking wait on the object. The blocked thread can be reactivated by another thread via the object's notify-method; the reactivated thread must re-apply for the lock before it may continue its execution. The method notifyAll, finally, notifies all threads blocked on the object.

2.2 Operational Semantics

A *local state* τ holds the values of the local variables of a method. A *local configuration* (α, τ, stm) of a thread executing within an object $\alpha \neq nil$ specifies, in addition to its local state τ, its point of execution represented by the statement *stm*. A *thread configuration* $\xi = (\alpha_0, \tau_0, stm_0) \ldots (\alpha_n, \tau_n, stm_n)$ is a stack of local configurations, representing the call chain of the thread. We write $\xi \circ (\alpha, \tau, stm)$ for pushing a new local configuration onto the stack.

An object is characterized by its *instance state* σ_{inst} which assigns values to the self-reference this and to instance variables. The initial states τ_{init} and

Table 2. Operational semantics of the monitor methods.

$$\frac{\begin{array}{c} m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\} \\ \beta = [\![e]\!]_{\mathcal{E}}^{\sigma(\alpha),\tau} \in \mathit{dom}(\sigma) \qquad \mathit{owns}(\xi \circ (\alpha, \tau, e.m(); \mathit{stm}), \beta) \end{array}}{\langle T \dot{\cup} \{\xi \circ (\alpha, \tau, e.m(); \mathit{stm})\}, \sigma\rangle \longrightarrow \langle T \dot{\cup} \{\xi \circ (\alpha, \tau, \mathit{stm}) \circ (\beta, \tau_{init}^{m,c}, \mathit{body}_{m,c})\}, \sigma\rangle} \ \textsc{Call}_{monitor}$$

$$\frac{\neg \mathit{owns}(T, \beta)}{\langle T \dot{\cup} \{\xi \circ (\alpha, \tau, \mathsf{receive}; \mathit{stm}) \circ (\beta, \tau', \mathsf{return}_{getlock})\}, \sigma\rangle \longrightarrow \langle T \dot{\cup} \{\xi \circ (\alpha, \tau, \mathit{stm})\}, \sigma\rangle} \ \textsc{Return}_{wait}$$

$$\frac{}{\begin{array}{l}\langle T \dot{\cup} \{\xi \circ (\alpha, \tau, !\mathsf{signal}; \mathit{stm})\} \dot{\cup} \{\xi' \circ (\alpha, \tau', ?\mathsf{signal}; \mathit{stm}')\}, \sigma\rangle \longrightarrow \\ \langle T \dot{\cup} \{\xi \circ (\alpha, \tau, \mathit{stm})\} \dot{\cup} \{\xi' \circ (\alpha, \tau', \mathit{stm}')\}, \sigma\rangle\end{array}} \ \textsc{Signal}$$

$$\frac{\mathit{wait}(T, \alpha) = \emptyset}{\langle T \dot{\cup} \{\xi \circ (\alpha, \tau, !\mathsf{signal}; \mathit{stm})\}, \sigma\rangle \longrightarrow \langle T \dot{\cup} \{\xi \circ (\alpha, \tau, \mathit{stm})\}, \sigma\rangle} \ \textsc{Signal}_{skip}$$

$$\frac{T' = \mathit{signal}(T, \alpha)}{\langle T \dot{\cup} \{\xi \circ (\alpha, \tau, !\mathsf{signal\_all}; \mathit{stm})\}, \sigma\rangle \longrightarrow \langle T' \dot{\cup} \{\xi \circ (\alpha, \tau, \mathit{stm})\}, \sigma\rangle} \ \textsc{SignalAll}$$

σ_{inst}^{init} assign initial values to all variables. A *global state* or *heap* σ maps each currently *existing* object, i.e., an object of its domain, to its instance state. A *global configuration* $\langle T, \sigma\rangle$ consists of a set T of thread configurations of the currently executing threads, together with a global state σ.

Expressions are evaluated with respect to an *instance local* state (σ_{inst}, τ); the base cases are $[\![u]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} = \tau(u)$, $[\![x]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} = \sigma_{inst}(x)$, and $[\![\mathsf{this}]\!]_{\mathcal{E}}^{\sigma_{inst},\tau} = \sigma_{inst}(\mathsf{this})$. The operational semantics is given as transitions between global configurations. A program's initial configuration $\langle T_0, \sigma_0\rangle$ satisfies $T_0 = \{(\alpha, \tau_{init}, \mathit{body}_{\mathsf{main},c})\}$ and $\sigma_0(\alpha) = \sigma_{inst}^{init}[\mathsf{this} \mapsto \alpha]$, where the domain of σ_0 is $\{\alpha\}$ and α is an instance of the main class c.

For the semantics of assignment, object and thread creation, and ordinary method invocation we refer to [4]. The rules of Table 2 handle *Java*$_{MT}$'s monitor methods wait, notify, and notifyAll, offering a typical monitor synchronization mechanism.

In all three cases the caller must own the lock of the callee object (cf. rule $\textsc{Call}_{monitor}$), as expressed by the predicate *owns*, defined below.

A thread can *block* itself on an object whose lock it owns by invoking the object's wait-method, thereby relinquishing the lock and placing itself into the object's wait set. Formally, the wait set $\mathit{wait}(T, \alpha)$ of an object is given as the set of all stacks in T with a top element of the form $(\alpha, \tau, ?\mathsf{signal}; \mathit{stm})$. After having "put itself on ice", the thread awaits notification by another thread which invokes the notify-method of the object. The !signal-statement in the notify-method thus reactivates a single thread waiting for notification on the given object (cf. rule

SIGNAL). Analogous to the wait set, the notified set $\mathit{notified}(T, \alpha)$ of α is the set of all stacks in T with top element of the form $(\alpha, \tau, \mathsf{return}_{getlock})$, i.e., threads which have been notified and trying to get hold of the lock again. According to rule RETURN$_{wait}$, the receiver can continue after notification in executing $\mathsf{return}_{getlock}$ only if the lock is free. Note that the notifier does not hand over the lock to the one being notified but continues to own it. This behavior is known as *signal-and-continue* monitor discipline [6].

If no threads are waiting on the object, the !signal of the notifier is without effect (cf. rule SIGNAL$_{skip}$). The notifyAll-method generalizes notify in that all waiting threads are notified via the !signal_all-broadcast (cf. rule SIGNALALL). The effect of this statement is given by setting $\mathit{signal}(T, \alpha)$ as $(T \setminus \mathit{wait}(T, \alpha)) \cup \{\xi \circ (\beta, \tau, \mathit{stm}) \mid \xi \circ (\beta, \tau, ?\mathsf{signal}; \mathit{stm}) \in \mathit{wait}(T, \alpha)\}$.

Using the wait and notified sets, we can now formalize the *owns* predicate: A thread ξ owns the lock of β iff ξ executes some synchronized method of β, but not its wait-method. Formally, $\mathit{owns}(T, \beta)$ is true iff there exists a thread $\xi \in T$ and a $(\beta, \tau, \mathit{stm}) \in \xi$ with stm synchronized and $\xi \notin \mathit{wait}(T, \beta) \cup \mathit{notified}(T, \beta)$. An invariant of the semantics is that at most one thread can own the lock of an object at a time.

3 The Assertion Language

Java$_{MT}$ does not allow qualified references to instance variables, to support a clean interface between between internal and external object behavior. To mirror this modularity, the assertion logic consists of a *local* and a *global* sublanguage. *Local* assertions are used to annotate methods in terms of their local variables and of the instance variables of the class to which they belong. *Global* assertions describe a whole system of objects and their communication structure and will be used in the cooperation test. In the assertion language we add the type Object as the supertype of all classes, and we introduce *logical variables* $z \in \mathit{LVar}$ different from all program variables. Logical variables are used for quantification and as free variables to represent local variables in the global assertion language. Expressions and assertions are interpreted relative to a logical environment ω, assigning values to logical variables.

Assertions are built using the usual constructs from predicate logic (cf. Table 3), where only the difference between the local and the global level deserves mention which concerns the form of quantification. On the local language, unrestricted quantification $\exists z.p$ is solely allowed for integer and boolean domains, but not for reference types, as for those types the range of quantification dynamically depends on the *global* state, something one cannot speak about on the local level. Nevertheless, one can assert the existence of objects on the local level, provided one is explicit about the domain of quantification, as in the restricted quantifications $\exists z{\in}e.p$ or $\exists z{\sqsubseteq}e.p$. The local assertion $\exists z{\in}e.p$ states that there is a value z in the sequence e, for which p holds; $\exists z{\sqsubseteq}e.p$ states the existence of a subsequence. Global assertions are evaluated in the context of a global state. Thus, unrestricted quantification is allowed for all types and ranges over the set of *existing* values. Qualified references $E.x$ may be used in the global

Table 3. Syntax of assertions.

$$\begin{array}{rcll}
exp_l & ::= & z \mid x \mid u \mid \mathsf{this} \mid \mathsf{null} \mid \mathsf{f}(exp_l, \ldots, exp_l) & e \in LExp \\
ass_l & ::= & exp_l \mid \neg ass_l \mid ass_l \wedge ass_l & \\
 & \mid & \exists z.\ ass_l \mid \exists z \in exp_l.\ ass_l \mid \exists z \sqsubseteq exp_l.\ ass_l & p \in LAss \\
 & & & \\
exp_g & ::= & z \mid \mathsf{null} \mid \mathsf{f}(exp_g, \ldots, exp_g) \mid exp_g.x & E \in GExp \\
ass_g & ::= & exp_g \mid \neg ass_g \mid ass_g \wedge ass_g \mid \exists z.\ ass_g & P \in GAss
\end{array}$$

language only. We write $[\![\_]\!]_{\mathcal{L}}$ and $[\![\_]\!]_{\mathcal{G}}$ for the semantic functions evaluating local and global assertions, and $\omega, \sigma_{inst}, \tau \models_{\mathcal{L}} p$ for $[\![p]\!]_{\mathcal{L}}^{\omega,\sigma_{inst},\tau} = true$, and $\models_{\mathcal{L}} p$ if p holds in all contexts; we use analogously $\models_{\mathcal{G}}$ for global assertions.

To express a local property p in the global assertion language, we define the lifting substitution $p[z/\mathsf{this}]$ by simultaneously replacing in p all occurrences of this by z, and transforming all occurrences of instance variables x into qualified references $z.x$, where z is assumed not to occur in p. For notational convenience we view the local variables occurring in the global assertion $p[z/\mathsf{this}]$ as logical variables. Formally, these local variables are replaced by fresh logical variables. We will write $P(z)$ for $p[z/\mathsf{this}]$, and similarly for expressions.

For technical convenience, in this paper we formulate verification conditions as standard Hoare-triples $\{\varphi\}\ stm\ \{\psi\}$. The statements of these Hoare-triples may also contain assignments involving qualified references as given by the global assertion language. The formal semantics is given by means of a weakest precondition calculus [8, 4].

4 The Proof System

The following section defines how to augment and annotate programs resulting in proof outlines, before Section 4.2 describes the proof method. The proof system accommodates for dynamic object creation, shared-variable concurrency, aliasing, method invocation, synchronization, and, especially, the reentrant monitors. Missing details can be found in [4].

4.1 Proof Outlines

The definition of a relatively complete proof system requires that we can encode the transition semantics of $Java_{MT}$ in the assertion language. As the assertion language can reason about the local and global states, only, we have to *augment* programs with fresh auxiliary variables to represent information about the control points and stack structures within the local and global states: Assignments $y := e$ can be extended to multiple assignments $y, \boldsymbol{y}_{aux} := e, \boldsymbol{e}_{aux}$ by inserting additional assignments to distinct auxiliary variables $\boldsymbol{y}_{aux}$. Additional auxiliary assignments can be inserted at any control point. Observations $\boldsymbol{y} := \boldsymbol{e}$ of communication and object creation stm are enclosed in *bracketed sections* $\langle stm; \boldsymbol{y} := \boldsymbol{e}\rangle$. Method calls and the reception of the return value is observed by $\langle e_0.m(\boldsymbol{e}); \boldsymbol{y}_1 := \boldsymbol{e}_1\rangle$ and $\langle \mathsf{receive}\, u_{ret}; \boldsymbol{y}_4 := \boldsymbol{e}_4\rangle$. Similarly for the callee, methods

```
...⟨e0.m(this,conf,thread,e);⟩ ⟨receive u_ret;⟩ ... // meth. call

sync m(caller,thread,u) {
       ⟨conf:=counter, counter:=counter+1, lock:=inc(lock)⟩ ...
       ⟨return e_ret; lock:=dec(lock)⟩ }

nsync start(caller,thread,caller_thread) {
       ⟨conf:=counter, counter:=counter+1, started:=true⟩ ...
       ⟨return;⟩}
```

Fig. 1. Augmentation and annotation: Synchronized method call, start.

```
nsync wait(caller,thread) {
      ⟨conf:=counter, counter:=counter+1,
       wait:=wait ∪ {lock}, lock:=free;⟩
      ⟨return_getlock; lock:=get(notified,thread),
       notified:=notified \ get(notified,thread);⟩ }

nsync notify(caller,thread) {
      ⟨conf:=counter, counter:=counter+1;⟩
      wait,notified:=notify(wait,notified);
      ⟨return;⟩ }

nsync notifyAll(caller,thread) {
      ⟨conf:=counter, counter:=counter+1;⟩
      notified:=notified ∪ wait, wait:=∅;
      ⟨return;⟩ }
```

Fig. 2. Augmentation and annotation: Signaling.

are extended to $m(\boldsymbol{u})\{\langle \boldsymbol{y}_2 := \boldsymbol{e}_2\rangle; stm; \langle \mathsf{return}\, u_{ret}; \boldsymbol{y}_3 := \boldsymbol{e}_3\rangle\}$. To be uniform, we will sometimes write $\langle ?m(\boldsymbol{u}); \boldsymbol{y} := \boldsymbol{e}\rangle$ to indicate that the assignment observes the reception of a method call. We require that the caller observation in a self-communication does not change the values of instance variables.

Bracketed sections do not influence the control flow of the original program but enforce a particular scheduling policy: Communication, sender, and receiver observations are executed in this order in a single computation step, i.e., they cannot be interleaved with other threads. Points which can be interleaved we call control points. At points between communication and its observation in bracketed sections, or at the beginning and at the end of a methods, no interleaving can take place; we call them *auxiliary points.*

Next we introduce a few auxiliary variables, built into all augmentations and used in the verification conditions. Their values are changed only as described in the following. The updating of the specific auxiliary variables for ordinary synchronized methods and for the start-method is illustrated in Figure 1. Non-synchronized methods are treated analogously except that they do not change the lock value; the start-method additionally handles thread creation.

Figure 2 shows the augmentation of the monitor methods. Note that we do not use the auxiliary statements !signal, !signal_all, and ?signal in the proof outlines and implement the monitore methods with the auxiliary variables wait and notified, instead, which represent the corresponding sets of the semantics.

An important point of the proof system is the identification of communicating objects and threads. We identify a thread by the object in which it has begun its execution. The identification is unique as an object's thread can be started only once. This identity is handed over from caller to callee as auxiliary formal parameter thread. For the start-method we use $\mathsf{caller\_thread}$ as additional formal parameter to store the identity of the caller thread. A local configuration, which represents the execution of a method, is identified by the object in which it executes together with the value of its auxiliary local variable conf storing a unique object-internal identifier. Its uniqueness is assured by the auxiliary instance variable $\mathsf{counter}$, incremented for each new local configuration in that object. The callee receives the "return address" as auxiliary formal parameter caller, given by the caller object together with the identity of the calling local configuration. The main-method is initially executed with the parameters $((nil, 0), \alpha_0)$, where α_0 is the initial object.

To capture mutual exclusion and the monitor discipline, the instance variable lock of type $\mathsf{Object} \times \mathsf{Int} + \mathsf{free}$, with initial value *free*, stores the identity of the thread that owns the lock, if any, together with the number of reentrant synchronized calls in the call chain. The semantics of incrementing the lock $[\![\mathsf{inc}(\mathsf{lock})]\!]_{\mathcal{E}}^{\sigma_{inst},\tau}$ is $(\tau(\mathsf{thread}), 0)$ for $\sigma_{inst}(\mathsf{lock}) = \mathit{free}$, and $(\alpha, n+1)$ for $\sigma_{inst}(\mathsf{lock}) = (\alpha, n)$. Decrementing $\mathsf{dec}(\mathsf{lock})$ is done inversely. The instance variables wait and $\mathsf{notified}$ of type $2^{\mathsf{Object} \times \mathsf{Int}}$, with initial value $\emptyset$, are the analogues of the *wait*- and *notified*-sets of the semantics and store the threads waiting at the monitor, respectively, those having been notified. Besides the thread identity, the number of reentrant synchronized calls is stored. In other words, the wait and notified sets remember the old lock-value prior to suspension which is restored when the thread becomes active again. The old value is given by $get(notified, \alpha)$ for a thread α, whose uniqueness is assured by the semantics. The value $notify(wait, notified)$ is the pair of the given sets with one element, chosen nondeterministically, moved from the wait into the notified set; if the wait set is empty, it is the identity function. Note that in the augmented wait-method both the waiting and the notified status of the executing thread are represented by a single control point. The two statuses can be distinguished by the values of the wait and $\mathsf{notified}$ variables. The boolean instance variable $\mathsf{started}$, finally, remembers whether the object's start-method has already been invoked. All auxiliary variables are initialized as usual, except the $\mathsf{started}$-variable of the initial object which gets the value true.

To specify invariant properties of the system, the augmented programs are *annotated* by attaching local assertions to each control and auxiliary point. We use the standard triple notation $\{p\}\ stm\ \{q\}$ and write $pre(stm)$ and $post(stm)$ to refer to the pre- and the post-condition of a statement. Besides that, for each class c, a local assertion I_c called *class invariant* specifies invariant properties of instances of c in terms of its instance variables. We require that the pre- and postconditions of whole method bodies, describing the instance state of the callee object directly before method call and after returning, respectively, are given by the class invariant.[1] Finally, the *global invariant* $GI \in GAss$ specifies properties

[1] Note that the callee configuration directly *after* invocation is described by the post-condition of the callee observation $\langle stm \rangle^{?call}$, which can be an arbitrary local assertion.

of communication between objects. As such, it should be invariant under object-internal computation. For that reason, we require that for all qualified references $E.x$ in GI with E of type c, all assignments to x in class c occur in bracketed sections of communication or object creation. Note that the global invariant is not affected by the object-internal monitor signaling mechanism. We require that in the annotation no free logical variables occur. An augmented and annotated program $prog'$ is called a *proof outline.*

4.2 Verification Conditions

The proof system formalizes a number of *verification conditions* which inductively ensure that for each *reachable* configuration the assertions at all current control points are satisfied, and that the global and the class invariants hold. The conditions are grouped, as usual, into initial conditions, local correctness, interference freedom, and a cooperation test. Note that the proof method is *modular* in that it allows for separate interference freedom and cooperation tests.

Arguing about two different local configurations makes it necessary to distinguish between their local variables, since these possibly have the same names; in such cases we rename the local variables in one of the local states. We use primed assertions p' to denote a given assertion p with every local variable u replaced by a fresh one u', and do so, correspondingly, for expressions.

4.2.1 Local Correctness. A proof outline is *locally correct*, if the properties of method instances as specified by the annotation are invariant under their own execution. For example, an assignment's precondition must imply its postcondition after execution. Besides that, invariance of the class invariant is required.

Definition 1 (Local correctness: Assignment). *A proof outline is* locally correct, *if for all assignments* $\{p_1\}\, \boldsymbol{y} := \boldsymbol{e}\, \{p_2\}$ *outside bracketed sections and all* c*,*

$$\models_{\mathcal{L}} \{p_1\} \quad \boldsymbol{y} := \boldsymbol{e} \quad \{p_2\} \tag{1}$$

$$\models_{\mathcal{L}} p_1 \rightarrow I_c \; . \tag{2}$$

The conditions for loops and conditional statements are similar. Note that we have no local verification conditions for observations of communication and object creation. The postconditions of such statements express *assumptions* about the communicated values. They will be verified in the *cooperation test.*

4.2.2 The Interference Freedom Test. Invariance of local assertions under computation steps in which they are not involved is assured by the *interference freedom test.* Since Java$_{MT}$ does not support qualified references to instance variables, we only have to deal with invariance under execution within the *same* object. Affecting only local variables, communication and object creation do not change the instance states of the executing objects. Thus we only have to cover invariance of assertions annotating control points over assignments, including those of bracketed sections. Assertions at auxiliary points do not have to be shown invariant. So let p be an assertion at a control point and $\boldsymbol{y} := \boldsymbol{e}$ an assignment in the same class. In the following we will prime local variables of the assertion to distinguish them from those of the assignment.

The assertion p has to be invariant under the assignment only if the assignment is executed independently of the control point annotated by p:

$$\begin{aligned}\mathsf{interferes}(p, \boldsymbol{y} := \boldsymbol{e}) \overset{def}{=} {} & \mathsf{thread} \neq \mathsf{thread}' \to \neg \mathsf{self\_start}(p, \boldsymbol{y} := \boldsymbol{e}) \wedge \\ & \mathsf{thread} = \mathsf{thread}' \to \mathsf{waits\_for\_ret}(p, \boldsymbol{y} := \boldsymbol{e}) .\end{aligned}$$

The definition distinguishes two cases: If the assertion and the assignment belong to *different* threads, interference freedom must be shown in any case except for the self-invocation of the start-method. If they belong to the *same* thread, the only assertions endangered are those at control points waiting for a return value earlier in the thread's stack. Invariance of a local configuration under its own execution, however, need not be considered and is excluded by requiring $\mathsf{conf} \neq \mathsf{conf}'$. Interference with the *matching* return statement in a self-communication neither needs to be considered, because communicating receive and return statements are executed simultaneously. This particular case is excluded by additionally requiring $\mathsf{caller} \neq (\mathsf{this}, \mathsf{conf}')$ for such assertion-assignment pairs.

Definition 2 (Interference freedom). *A proof outline is interference free, if for all classes c, assignments $\{p\}\boldsymbol{y} := \boldsymbol{e}$, and assertions q at control points in c,*

$$\models_{\mathcal{L}} \{p \wedge q' \wedge \mathsf{interferes}(q, \boldsymbol{y} := \boldsymbol{e})\} \quad \boldsymbol{y} := \boldsymbol{e} \quad \{q'\} . \tag{3}$$

4.2.3 The Cooperation Test. Whereas the interference freedom test assures invariance of assertions under steps in which they are not involved, the *cooperation test* deals with inductivity for communicating partners, assuring that the global invariant and the preconditions of the involved bracketed sections imply their postconditions after the joint step. Additionally, the assertions at the auxiliary points must hold immediately after communication. The global invariant may refer only to auxiliary instance variables which are changed in bracketed sections. Consequently, it is automatically invariant under the execution of non-communicating statements. For bracketed sections of communication and object creation, however, the invariance must be shown as part of the cooperation test.

In the following we define the cooperation test for method call. Since different objects may be involved, the cooperation test is formulated in the global assertion language. Local properties are expressed in the global language using the lifting substitution. To avoid name clashes between local variables of the partners, we rename those of the callee by priming them.

Let z and z' be logical variables representing the caller, respectively, the callee object. We assume the global invariant and the preconditions of the communicating statements to hold prior to communication. For method invocation, the precondition of the callee is its class invariant, as defined in the annotation. That the assertions indeed represent communicating partners and that the communication is enabled is captured in the assertion comm: In case of a synchronized method invocation, the lock of the callee object has to be free or owned by the caller, which is expressed by $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$, where thread is the caller thread and $\mathit{thread}(\alpha, n) = \alpha$. For the invocation of the monitor methods the executing thread must hold the lock.

Let the function $\mathit{Init} : \mathit{Var} \rightarrow \mathit{Val}_{\mathit{nil}}$ assign to each variable its initial value.

Definition 3 (Cooperation test: Method invocation). *A proof outline satisfies the cooperation test for method invocation, if for all statements of the form* $\{p_1\}\langle e_0.m(\boldsymbol{e}); \{p_2\}\boldsymbol{y}_1 := \boldsymbol{e}_1\rangle\{p_3\}$ *in class* c *with* e_0 *of type* c'*, where method* m *of* c' *has body* $\{q_1\}\langle ?m(\boldsymbol{u}); \{q_2\}\boldsymbol{y}_2 := \boldsymbol{e}_2\rangle; \{q_3\}\mathit{stm}$ *and local variables* $\boldsymbol{v}$ *except the formal parameters:*

$$\models_{\mathcal{G}} \quad \begin{array}{c}\{GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{comm}\} \\ \boldsymbol{u}', \boldsymbol{v}' := \boldsymbol{E}(z), \mathsf{Init}(\boldsymbol{v}) \\ \{P_2(z) \wedge Q_2'(z')\}\end{array}$$

$$\models_{\mathcal{G}} \quad \begin{array}{c}\{GI \wedge P_1(z) \wedge Q_1'(z') \wedge \mathsf{comm}\} \\ \boldsymbol{u}', \boldsymbol{v}' := \boldsymbol{E}(z), \mathsf{Init}(\boldsymbol{v}); \quad z.\boldsymbol{y}_1 := \boldsymbol{E}_1(z); \quad z'.\boldsymbol{y}_2' := \boldsymbol{E}_2'(z') \\ \{GI \wedge P_3(z) \wedge Q_3'(z')\},\end{array}$$

where $z \in \mathit{LVar}^c$ *and* $z' \in \mathit{LVar}^{c'}$ *are distinct and fresh;* comm *is given by* $E_0(z) = z' \wedge z \neq \mathsf{nil} \wedge z' \neq \mathsf{nil} \wedge \mathsf{synch}$*, and with* synch *defined as*

- true *for* $m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ *non-synchronized,*
- $z'.\mathsf{lock} = \mathsf{free} \vee \mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ *for* $m \notin \{\mathsf{start}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$ *synchronized,*
- $\mathsf{thread}(z'.\mathsf{lock}) = \mathsf{thread}$ *for* $m \in \{\mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}$*, and*
- $\neg z'.\mathsf{started}$ *for* $m = \mathsf{start}$.

For $m = \mathsf{start}$*, the conditions must hold with additionally* synch *as* $z'.\mathsf{started}$*, where* $q_2 = q_3 = \mathsf{true}$ *and without* $\boldsymbol{u}', \boldsymbol{v}' := \boldsymbol{E}(z), \mathsf{Init}(\boldsymbol{v})$ *and* $z'.\boldsymbol{y}_2' := \boldsymbol{E}_2'(z')$.

The first verification condition justifies the assertions at the auxiliary points after parameter passing, whereas the second verification condition justifies the postconditions of the bracketed sections and invariance of the global invariant.

For returning from a method and for object creation, there are similar conditions. Here we remark only that returning from the wait-method assumes that the thread has been notified and that the lock of the given object is free; in this case, synch is defined by $z'.\mathsf{lock} = \mathsf{free} \wedge \mathsf{thread}' \in z'.\mathsf{notified}$ (with $\alpha \in \mathit{notified}$ iff there is a $(\alpha, n) \in \mathit{notified}$ for some n).

The verification conditions presented above give rise to a *sound* and *relative complete* proof system. This means if all the verification conditions are valid, then at each reachable point all assertions hold, and conversely, if a program satisfies the requirements asserted in its proof outline, then this is indeed provable, i.e., then there exists a proof outline which can be shown to hold and which implies the given one. For the exact (standard) formulation of soundness and completeness and their proofs, we refer to the technical report [4].

```
1        $...\{owns(thread, lock)\}$
2  ⟨this.wait(this,conf,thread)⟩;
3        $\{\neg owns(thread, lock) \wedge (this, conf) = getcaller(x, thread)\}$
4  ⟨receive⟩
5        $\{owns(thread, lock)\} \ldots$

7  $\{true\}$  nsync wait (caller,thread) {
8        $\{owns(thread, lock)\}$
9    ⟨conf:=counter, counter:=counter + 1, lock:=free,
10    wait:=wait ∪ {lock}, x:=x ∘ (thread,caller)⟩;
11       $\{\neg owns(thread, lock) \wedge caller = getcaller(x, thread)\}$
12   ⟨return$_{getlock}$;
13       $\{lock = free \wedge caller = getcaller(x, thread) \wedge thread \in notified\}$
14    lock:=get(notified,thread),
15    notified:=notified \ get(notified,thread)⟩ } $\{true\}$
```

Fig. 3. Example.

4.2.4 Example.

We conclude this section with an example shown in Figure 3 which presents an annotation of the wait-method and its self-invocation. The annotation expresses basic properties of the lock ownership. The history of all invocations of the wait method is recorded in the auxiliary instance variable x of type $\mathsf{list}(\mathsf{Object} \times (\mathsf{Object} \times \mathsf{Int}))$. The operation $getcaller(x, thread)$ returns $caller$ where $(thread, caller)$ is the last element in x with first component $thread$. We use $\mathsf{owns}(\mathsf{thread}, \mathsf{lock})$ as shorthand for $\mathsf{thread} = \mathsf{thread}(\mathsf{lock})$.

The proof uses the interference freedom test and the cooperation test; we start with the latter. The postconditions (3) and (9) of the method invocation at (2) and its observation at (8) is justified in the cooperation test as the postcondition of the parameter passing followed by the observation; note that (7) follows directly from (1) by parameter passing. Remember that we rename the local variables of the callee. The cooperation test for returning from the wait-method, i.e., the simultaneous execution of (4) and (10), additionally imports the information described by the assertion synch, namely that the lock is free and the executing thread is already notified.

To show interference freedom, we proceed by case analysis. For interference between different threads, assume $\mathsf{thread} \neq \mathsf{thread}'$. For example, the assertion at (1) is invariant under the execution of (8) by a different thread, because the lock can be owned by at most one thread. Formally, the conjunction of (1) and (7) gives us $\mathsf{owns}(\mathsf{thread}', \mathsf{lock}) \wedge \mathsf{owns}(\mathsf{thread}, \mathsf{lock})$ contradicting our assumption $\mathsf{thread} \neq \mathsf{thread}'$. The other cases follow similarly from the assumption.

More interesting is interference with respect to one thread, where we only need to consider invariance of the primed assertion at (3) over the assignments at (8) and (12). For (8), it suffices to observe that the assertions (3) and (7) lead to a contradiction with the assumption $\mathsf{thread} = \mathsf{thread}'$. The most interesting case is the invariance of the assertion (3) under the execution of (12). The information about the auxiliary variable x and the assumption that we deal with a single thread implies $\mathsf{caller} = (\mathsf{this}, \mathsf{conf}')$. This information contradicts the $\mathsf{interleavable}$ predicate of the interference freedom test, which excludes the situation of a matching return-receive statements in a self-communication.

5 Conclusion

This paper presents the first sound and complete assertional proof method for a multithreaded sublanguage of *Java* including its monitor discipline. It extends earlier work [5] by integrating *Java*'s wait and notify constructs into the proof system and by moving towards a more compositional formulation of the identification mechanism for threads, corresponding to the compositional semantics in [3]. Moreover, this particular extension shows how to incorporate further control mechanism by means of auxiliary variables which describe the corresponding flow of control. Based on the proof theory presented here, we have developed a front-end tool *Verger* which automatically extends programs with the built-in augmentation and generates the verification conditions for the theorem prover *PVS*. In [4], we explore the tool on a few examples, and present an extension of the proof theory to show absence of deadlock.

Related Work. As far as proof systems and verification support for object-oriented programs is concerned, research has mostly concentrated on *sequential* languages. For instance, the LOOP-project [14, 17] develops methods and tools for the verification of sequential object-oriented languages, based on coalgebras and using proof *PVS* and *Isabelle/HOL*. Poetzsch-Heffter and Müller [20] develop a Hoare-style programming logic presented in sequent formulation for a sequential kernel of *Java*, featuring interfaces, subtyping, and inheritance. Translating the operational and the axiomatic semantics into the HOL theorem prover allows a computer assisted soundness proof. The work [21] uses a modification of the *object constraint language* OCL as assertional language to annotate UML class diagrams and to generate proof conditions for *Java*-programs. In [23] a large subset of *JavaCard*, including exception handling, is formalized in *Isabelle/HOL*, and its soundness and completeness is shown within the theorem prover. [18] presents an exectuable formalization of a simplified JVM within the theorem prover ACL2. The work in [2] presents a Hoare-style proof-system for a sequential object-oriented calculus [1]. The language features heap-allocated objects (but no classes), side-effects and aliasing, and its type system supports subtyping. Furthermore, the language allows nested statically let-bound variables, which requires a more complex semantical treatment for variables based on closures, and ultimately renders their proof-system incomplete. Its assertion language is presented as an extension of the object calculus' language of type and analogously, the proof system extends the type derivation system. The close connection of types and specifications in the presentation is exploited in [22] for the generation of verification conditions. A survey about *monitors* in general, including proof-rules for various monitor semantics, can be found in [7].

The *extended static checking* approach [9, 15] occupies a middle-ground between verification and static analysis. Based on an intermediate guarded command language, the ESC-tool statically tries to detect (amongst other static properties) programming errors typical in a multithreaded setting such as synchronization errors and race conditions. In this direction, [10] presents a thread-modular checking approach based on assume-guarantee reasoning and implemented in the *Calvin*-tool.

Future Work. We plan to extend Java$_{MT}$ by further constructs, like exceptions, inheritance, and subtyping. To deal with subtyping on the logical level requires a notion of behavioral subtyping.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer, 1996.
2. M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *Proceedings of TAPSOFT '97*, LNCS 1214, pages 682–696, Lille, France, Apr. 1997. Springer-Verlag. An extended version of this paper appeared as SRC Research Report 161 (September 1998).
3. E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. A compositional operational semantics for Java$_{MT}$. In N. Derschowitz, editor, *International Symposium on Verification (Theory and Practice)*, LNCS 2772. Springer-Verlag, 2003. To appear. A preliminary version appeared as Technical Report TR-ST-02-2, May 2002.
4. E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. A Hoare logic for monitors in Java. Techical report TR-ST-03-1, Lehrstuhl für Software-Technologie, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Apr. 2003.
5. E. Ábrahám-Mumm, F. S. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In M. Nielsen and U. H. Engberg, editors, *Proceedings of FoSSaCS 2002*, LNCS 2303, pages 4–20. Springer-Verlag, Apr. 2002. A longer version, including the proofs for soundness and completeness, appeared as Technical Report TR-ST-02-1, March 2002.
6. G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.
7. P. A. Buhr, M. Fortier, and M. H. Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, Mar. 1995.
8. F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Proceedings of FoSSaCS '99*, LNCS 1578, pages 135–156. Springer-Verlag, 1999.
9. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. SRC Technical Note 159, Compaq, Dec. 1998.
10. C. Flanagan, S. Qadeer, and S. Seshia. A modular checker for multithreaded programs. In E. Brinksma and K. G. Larsen, editors, *Proceedings of CAV '02*, LNCS 2404, pages 180–194. Springer-Verlag, 2002.
11. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. in Applied Mathematics*, volume 19, pages 19–32, 1967.
12. J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification.* Addison-Wesley, 1996.
13. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.
14. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, 2001.
15. K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. SRC Technical Note 1999-002, Compaq, May 1999.
16. G. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.

17. The LOOP project: Formal methods for object-oriented systems. http://www.cs.kun.nl/~bart/LOOP/, 2001.
18. J. S. Moore and G. M. Porter. An executable formal Java Virtual Machine thread model. In *Proceedings of the 2001 JVM Usenix Symposium in Monterey, California*, 2001.
19. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
20. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. Swierstra, editor, *Programming Languages and Systems*, LNCS 1576, pages 162–176. Springer, 1999.
21. B. Reus, R. Hennicker, and M. Wirsing. A Hoare calculus for verifying Java realizations of OCL-constrained design models. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering*, LNCS 2029, pages 300–316. Springer-Verlag, 2001.
22. F. Tang and M. Hofmann. Generation of verification conditions for Abadi and Leino's logic of objects (extended abstract). In *Proceedings of the 9th International Workshop on Foundations of Object-Oriented Languages (FOOL'02)*, 2002. A longer version is available as LFCS technical report.
23. D. von Oheimb and T. Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In L.-H. Eriksson and P.-A. Lindsay, editors, *Proceedings of Formal Methods Europe: Formal Methods – Getting IT Right (FME'02)*, LNCS 2391, pages 89–105. Springer-Verlag, 2002.
24. J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling With Uml.* Object Technology Series. Addison-Wesley, 1999.

Proof Scores in the OTS/CafeOBJ Method

Kazuhiro Ogata[1,2] and Kokichi Futatsugi[2]

[1] NEC Software Hokuriku, Ltd.
ogatak@acm.org
[2] Japan Advanced Institute of Science and Technology (JAIST)
kokichi@jaist.ac.jp

Abstract. A way to write proof scores showing that distributed systems have invariant properties in algebraic specification languages is described, which has been devised through several case studies. The way makes it possible to divide a formula stating an invariant property under discussion into reasonably small ones, each of which is proved by writing proof scores individually. This relieves the load to reduce logical formulas and can decrease the number of subcases into which the case is split in case analysis.

Keywords: Algebraic specification, CafeOBJ, observational transition system, proof scores, the NSLPK authentication protocol, verification.

1 Introduction

Equations are the most basic logical formulas and equational reasoning is the most fundamental way of reasoning[1], which can moderate the difficulties of proofs that might otherwise become too hard to understand. Algebraic specification languages make it possible to describe systems in terms of equations and verify that systems have properties by means of equational reasoning. Writing proofs, or proof scores in algebraic specification languages has been mainly promoted by researchers of the OBJ community[2].

We have been successfully applying such algebraic techniques to modeling, specification and verification of distributed systems such as distributed mutual exclusion algorithms[3, 4] and security protocols[5, 6]. In our method called the OTS/CafeOBJ method, systems are modeled as observational transition systems, or OTSs, which are described in CafeOBJ[7, 8], an algebraic specification language. The CafeOBJ description of OTSs can be regarded as restricted behavioral specification[9]. We verify that OTSs, which are models of systems, have properties by writing proof scores in CafeOBJ.

In this paper, we describe a way to write proof scores showing that distributed systems have invariant properties, which are most basic and important among various kinds of properties because proofs of other kinds of properties often need invariants. We have devised the way through several case studies[3–6]. The way makes it possible to divide a formula stating an invariant property under discussion into reasonably small ones, each of which is proved by writing proof scores individually. This relieves the load to reduce logical formulas and can decrease the number of subcases into which the case is split in case analysis. The proofs of the small formulas may depend on each other in the sense that the proof of one uses some other to strengthen inductive hypotheses and vice versa.

The rest of the paper is organized as follows. Section 2 mentions CafeOBJ and OTSs. Section 3 describes compositional proofs of invariants. A way of writing proof scores based on the compositional proofs of invariants is described in Sect. 4. Section 5 uses the NSLPK authentication protocol[10, 11] as an example to demonstrate how to write proof scores. Section 6 discusses the advantages of our method and concludes the paper.

2 Preliminaries

2.1 CafeOBJ in a Nutshell

CafeOBJ[7, 8] can be used to specify abstract machines as well as abstract data types. A visible sort denotes an abstract data type, while a hidden sort the state space of an abstract machine. There are two kinds of operators to hidden sorts: action and observation operators. An action operator can change states of an abstract machine. Only observation operators can be used to observe the inside of an abstract machine. An action operator is basically specified with equations by describing how the value of each observation operator changes. Declarations of observation and action operators start with `bop` or `bops`, and those of other operators with `op` or `ops`. Declarations of equations start with `eq`, and those of conditional ones with `ceq`. The CafeOBJ system rewrites a given term by regarding equations as left-to-right rewrite rules.

2.2 Observational Transition Systems

We assume that there exists a universal state space called Υ. We also suppose that each data type used has been defined beforehand, including the equivalence between two data values v_1, v_2 denoted by $v_1 = v_2$. A system is modeled by observing, from the outside of each state of Υ, only quantities that are relevant to the system and how to change the quantities by state transition. An OTS (observational transition system) can be used to model a system in this way. An OTS $\mathcal{S} = \langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ consists of:

- $\mathcal{O}$: A set of observable values. Each $o \in \mathcal{O}$ is a function $o : \Upsilon \to D$, where D is a data type and may be different for each observable value. Given an OTS $\mathcal{S}$ and two states $v_1, v_2 \in \Upsilon$, the equivalence between two states, denoted by $v_1 =_{\mathcal{S}} v_2$, w.r.t. $\mathcal{S}$ is defined as $v_1 =_{\mathcal{S}} v_2 \overset{\text{def}}{=} \forall o \in \mathcal{O}.o(v_1) = o(v_2)$.
- $\mathcal{I}$: The set of initial states such that $\mathcal{I} \subset \Upsilon$.
- $\mathcal{T}$: A set of conditional transition rules. Each $\tau \in \mathcal{T}$ is a function $\tau : \Upsilon/{=_{\mathcal{S}}} \to \Upsilon/{=_{\mathcal{S}}}$ on equivalence classes of Υ w.r.t. $=_{\mathcal{S}}$. Let $\tau(v)$ be the representative element of $\tau([v])$ for each $v \in \Upsilon$ and it is called *the successor state* of v w.r.t. τ. The condition c_τ for a transition rule $\tau \in \mathcal{T}$, which is a predicate on states, is called *the effective condition*. The effective condition is supposed to satisfy the following requirement: given a state $v \in \Upsilon$, if c_τ is false in v, namely τ is not *effective* in v, then $v =_{\mathcal{S}} \tau(v)$.

An OTS is described in CafeOBJ. Observable values are denoted by CafeOBJ observations, and transition rules by CafeOBJ actions.

Multiple similar observable values and transition rules may be indexed. Generally, observable values and transition rules are denoted by $o_{i_1,\ldots,i_m}$ and $\tau_{j_1,\ldots,j_n}$, respectively, provided that $m, n \geq 0$ and we assume that there exist data types D_k such that $k \in D_k$ ($k = i_1, \ldots, i_m, j_1, \ldots, j_n$). For example, an integer array a possessed by a process p may be denoted by an observable value a_p, and the increment of the ith element of the array may be denoted by a transition rule $\mathit{inc}\text{-}a_{p,i}$.

An execution of $\mathcal{S}$ is an infinite sequence $\upsilon_0, \upsilon_1, \ldots$ of states satisfying[1]:

- *Initiation*: $\upsilon_0 \in \mathcal{I}$.
- *Consecution*: For each $i \in \{0, 1, \ldots\}$, $\upsilon_{i+1} =_{\mathcal{S}} \tau(\upsilon_i)$ for some $\tau \in \mathcal{T}$.

A state is called *reachable* w.r.t. $\mathcal{S}$ iff it appears in an execution of $\mathcal{S}$. Let $\mathcal{R}_{\mathcal{S}}$ be the set of all the reachable states w.r.t. $\mathcal{S}$.

All properties considered in this paper are invariants[2], which are defined as follows:

$$\mathsf{invariant}\ p \overset{\text{def}}{=} (\forall \upsilon \in \mathcal{I}.\, p(\upsilon)) \wedge (\forall \upsilon \in \mathcal{R}_{\mathcal{S}}.\forall \tau \in \mathcal{T}.(p(\upsilon) \Rightarrow p(\tau(\upsilon))))\,,$$

which means that the predicate p is true in any reachable state of $\mathcal{S}$. Let $\boldsymbol{x}$ be all free variables except for one for states in p. We suppose that $\mathsf{invariant}\, p$ is interpreted as $\forall \boldsymbol{x}.(\mathsf{invariant}\, p)$ in this paper.

2.3 Description of OTSs in CafeOBJ

An OTS $\mathcal{S}$ is described in CafeOBJ. The universal state space Υ is denoted by a hidden sort, say H. An observable value $o_{i_1,\ldots,i_m} \in \mathcal{O}$ is denoted by a CafeOBJ observation operator. We assume that the data types D_k ($k = i_1, \ldots, i_m$) and D are described in initial algebra and there exist visible sorts V_k ($k = i_1, \ldots, i_m$) and V corresponding to the data types. The CafeOBJ observation operator denoting $o_{i_1,\ldots,i_m}$ is declared as follows:

bop o : $H\ V_{i_1}\ \ldots\ V_{i_m}$ -> V .

Any initial state in $\mathcal{I}$ is denoted by a constant (an operator with no arguments), say *init*, which is declared as follows:

op *init* : -> H

Suppose that the initial value of $o_{i_1,\ldots,i_m}$ is $f(i_1, \ldots, i_m)$. This is expressed by the following equation:

[1] If we want to discuss liveness properties, an execution of $\mathcal{S}$ should also satisfy *Fairness*: for each $\tau \in \mathcal{T}$, there exist an infinite number of indexes $i \in \{0, 1, \ldots\}$ such that $\upsilon_{i+1} =_{\mathcal{S}} \tau(\upsilon_i)$.

[2] In addition to invariant properties, there are unless, stable, ensures and leads-to properties, which are inspired by UNITY[12]. The way to write proof scores described in this paper can also be applied to unless, stable, ensures properties.

$\texttt{eq}\ o(init, X_{i_1}, \ldots, X_{i_m}) = f(X_{i_1}, \ldots, X_{i_m})\ .$

$X_k\ (k = i_1, \ldots, i_m)$ is a CafeOBJ variable for V_k and $f(X_{i_1}, \ldots, X_{i_m})$ is a term denoting $f(i_1, \ldots, i_m)$.

A transition rule $\tau_{j_1,\ldots,j_n} \in \mathcal{T}$ is denoted by a CafeOBJ action operator. We assume that the data types $D_k\ (k = j_1, \ldots, j_n)$ are described in initial algebra and there exist visible sorts $V_k\ (k = j_1, \ldots, j_n)$ corresponding to the data types. The CafeOBJ action operator denoting $\tau_{j_1,\ldots,j_n}$ is declared as follows:

$\texttt{bop}\ a : H\ V_{j_1}\ \ldots\ V_{j_n}\ \texttt{->}\ H\ .$

If $\tau_{j_1,\ldots,j_n}$ is applied in a state in which it is effective, the value of $o_{i_1,\ldots,i_m}$ may be changed, which can be described in CafeOBJ generally as follows:

$\texttt{ceq}\ o(a(W, X_{j_1}, \ldots, X_{j_n}), X_{i_1}, \ldots, X_{i_m}) = e\text{-}a(W, X_{j_1}, \ldots, X_{j_n}, X_{i_1}, \ldots, X_{i_m})$
$\quad\texttt{if}\ c\text{-}a(W, X_{j_1}, \ldots, X_{j_n})\ .$

W is a CafeOBJ variable for H and $X_k\ (k = j_1, \ldots, i_n)$ is a CafeOBJ variable for V_k. $a(W, X_{j_1}, \ldots, X_{j_n})$ denotes the successor state of W w.r.t. $\tau_{j_1,\ldots,j_n}$. $e\text{-}a(W, X_{j_1}, \ldots, X_{j_n}, X_{i_1}, \ldots, X_{i_m})$ denotes the value of $o_{i_1,\ldots,i_m}$ in the successor state. $c\text{-}a(W, X_{j_1}, \ldots, X_{j_n})$ denotes the effective condition $c_{\tau_{j_1,\ldots,j_n}}$ of $\tau_{j_1,\ldots,j_n}$.

If $\tau_{j_1,\ldots,j_n}$ is applied in a state in which it is not effective, the value of any observable value is not changed. Therefore all we have to do is to declare the following equation:

$\texttt{ceq}\ a(W, X_{j_1}, \ldots, X_{j_n}) = W\ \texttt{if not}\ c\text{-}a(W, X_{j_1}, \ldots, X_{j_n})\ .$

If the value of $o_{i_1,\ldots,i_m}$ is not affected by applying $\tau_{j_1,\ldots,j_n}$ in any state (regardless of the truth value of $c_{\tau_{j_1,\ldots,j_n}}$), the following equation may be declared:

$\texttt{eq}\ o(a(W, X_{j_1}, \ldots, X_{j_n}), X_{i_1}, \ldots, X_{i_m}) = o(W, X_{i_1}, \ldots, X_{i_m})\ .$

3 Compositional Proofs of Invariants

Suppose that we prove that a system has an invariant property. The system is first modeled as an OTS, which is described in CafeOBJ. Let H be the hidden sort denoting the state space Υ, and let the invariant be invariant $pred_1(s, \boldsymbol{x}_1)$, where s is a free variable for states and $\boldsymbol{x}_1$ the other free variables. It is often impossible to prove invariant $pred_1(s, \boldsymbol{x}_1)$ alone. Suppose that it is possible to prove that $pred_1(s, \boldsymbol{x}_1)$, together with $n - 1$ other predicates, is invariant to the OTS. Let the $n - 1$ predicates be $pred_2(s, \boldsymbol{x}_2), \ldots, pred_n(s, \boldsymbol{x}_n)$. That is, we prove invariant $(pred_1(s, \boldsymbol{x}_1) \wedge \ldots \wedge pred_n(s, \boldsymbol{x}_n))$, instead of the original invariant, from which the original invariant can be deduced. Let $pred(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ be $pred_1(s, \boldsymbol{x}_1) \wedge \ldots \wedge pred_n(s, \boldsymbol{x}_n)$.

Although sometimes invariants may be proved by reduction and/or case analysis only, we often need to use induction, especially induction on the number of transition rules applied or executed.

Suppose that invariant $pred(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ is proved by induction on the number of transition rules applied. Let us consider an inductive case in which it is shown that any transition rule denoted by a CafeOBJ action operator a preserves $\mathit{pred}(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$. To this end, it is sufficient to show this formula:

$$\mathit{pred}(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \Rightarrow \mathit{pred}(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \tag{1}$$

for any $s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n, \boldsymbol{y}$, where $\boldsymbol{y}$ is the arguments of the CafeOBJ action operator except for s. It is often the case that we cannot prove the formula as it is because the inductive hypothesis $\mathit{pred}(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ is too weak. Then, we can strengthen the inductive hypothesis by adding a formula, say *SIH*, of the following form:

$$\mathit{pred}(s, \boldsymbol{t}_1^1, \ldots, \boldsymbol{t}_n^1) \wedge \ldots \wedge \mathit{pred}(s, \boldsymbol{t}_1^m, \ldots, \boldsymbol{t}_n^m),$$

where $\boldsymbol{t}_1^i, \ldots, \boldsymbol{t}_n^i$ $(i = 1, \ldots, m)$ are lists of terms. Then, the proof of (1) can be replaced with the proof of the following formula:

$$(\mathit{SIH} \wedge \mathit{pred}(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)) \Rightarrow \mathit{pred}(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n).$$

This formula can be proved compositionally. The proof of the formula is equivalent to the proofs of the following n formulas:

$$\begin{gathered}(\mathit{SIH} \wedge \mathit{pred}(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)) \Rightarrow \mathit{pred}_1(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_1), \\ \vdots \\ (\mathit{SIH} \wedge \mathit{pred}(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)) \Rightarrow \mathit{pred}_n(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_n).\end{gathered} \tag{2}$$

Moreover, it suffices to prove the following n formulas, if possible, instead of the above n formulas:

$$\begin{gathered}\mathit{pred}_1(s, \boldsymbol{x}_1) \Rightarrow \mathit{pred}_1(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_1), \\ \vdots \\ \mathit{pred}_n(s, \boldsymbol{x}_n) \Rightarrow \mathit{pred}_n(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_n),\end{gathered} \tag{3}$$

because the ith formula of (2) can be deduced from the ith formula of (3), where $1 \leq i \leq n$.

Some of (3) cannot be proved as they are because their inductive hypotheses are too weak. Let $\mathit{pred}_i(s, \boldsymbol{x}_i) \Rightarrow \mathit{pred}_i(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_i)$, where $1 \leq i \leq n$, be one of such formulas. Suppose that $\mathit{pred}_j(s, \boldsymbol{u}_j)$, where $1 \leq j \leq n$ and $\boldsymbol{u}_j$ is $\boldsymbol{x}_j$, $\boldsymbol{t}_j^1$, ..., or $\boldsymbol{t}_j^m$, can be used to strengthen the inductive hypothesis $\mathit{pred}_i(s, \boldsymbol{x}_i)$ in order to prove the formula. The proof of the formula can be replaced with the proof of the following formula[3]:

$$(\mathit{pred}_j(s, \boldsymbol{u}_j) \wedge \mathit{pred}_i(s, \boldsymbol{x}_i)) \Rightarrow \mathit{pred}_i(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_i),$$

because the ith formula of (2) can be deduced from this formula. Generally what strengthens the inductive hypothesis can be $\mathit{pred}_{j_1}(s, \boldsymbol{u}_{j_1}) \wedge \ldots \wedge \mathit{pred}_{j_k}(s, \boldsymbol{u}_{j_k})$,

[3] If invariant $\mathit{pred}_j(s, \boldsymbol{x}_j)$ has been proved independent of invariant $\mathit{pred}_i(s, \boldsymbol{x}_i)$, the proof can also be replaced with the proof of the following:

$$(\mathit{pred}_j(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{u}_j) \wedge \mathit{pred}_i(s, \boldsymbol{x}_i)) \Rightarrow \mathit{pred}_i(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_i).$$

In this case, the jth invariant is used as usual lemma for the proof of the ith invariant.

where $1 \leq j_1, \ldots, j_k \leq n$ and $\boldsymbol{u}_j$ ($j = j_1, \ldots, j_k$) is $\boldsymbol{x}_j$, $\boldsymbol{t}_j^1$, ..., or $\boldsymbol{t}_j^m$. Let SIH_i be this formula to strengthen the inductive hypothesis $pred_i(s, \boldsymbol{x}_i)$. Then, the proof of the ith formula of (3) can be replaced with the proof of the following:

$$(SIH_i \wedge pred_i(s, \boldsymbol{x}_i)) \Rightarrow pred_i(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_i)\,. \tag{4}$$

Moreover, we may have to split the case into multiple subcases in order to prove (4). Suppose that the case is split into l subcases. The l subcases are denoted by l formulas $case_1^i, \ldots, case_l^i$, which should satisfy the following:

$$(case_1^i \vee \ldots \vee case_l^i) = \mathrm{true}.$$

Then, the proof of (4) can be replaced with the proofs of the following l formulas:

$$\begin{gathered}(case_1^i \wedge SIH_i \wedge pred_i(s, \boldsymbol{x}_i)) \Rightarrow pred_i(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_i)\,,\\ \vdots \\ (case_l^i \wedge SIH_i \wedge pred_i(s, \boldsymbol{x}_i)) \Rightarrow pred_i(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_i)\,.\end{gathered} \tag{5}$$

SIH_i may not be needed for some subcases.

From what has been discussed, it follows that the n invariants can be proved compositionally even if they depend on each other, namely that the ith invariant is used to strengthen inductive hypotheses for the proof of the jth invariant and vice versa. The original invariant invariant $pred_1(s, \boldsymbol{x}_1)$, which we would like to prove, may be divided into multiple invariants. Proof scores in the OTS/CafeOBJ method are based on what has been discussed, especially (5), and therefore, we can write proof scores of the n invariants individually.

4 Proof Scores of Invariants

Let us consider that we write proof scores of the n invariants discussed in the previous section. We first write a module, say `INV`, where $pred_i(s, \boldsymbol{x}_i)$ ($i = 1, \ldots, n$) is expressed as a CafeOBJ term as follows:

```
op inv_1 : H V_1 -> Bool
...
op inv_n : H V_n -> Bool
eq inv_1(W, X_1) = pred_1(W, X_1) .
...
eq inv_n(W, X_n) = pred_n(W, X_n) .
```

$\boldsymbol{V}_i$ ($i = 1, \ldots, n$) is a list of visible sorts corresponding to $\boldsymbol{x}_i$, W is a CafeOBJ variable for the hidden sort H and $\boldsymbol{X}_i$ ($i = 1, \ldots, n$) is a list of CafeOBJ variables for $\boldsymbol{V}_i$. The term $pred_i(W, \boldsymbol{X}_i)$ ($i = 1, \ldots, n$) denotes $pred_i(s, \boldsymbol{x}_i)$.

In the module, we also declare constants $\mathbf{x}_i$ ($i = 1, \ldots, n$) for $\boldsymbol{V}_i$. In proof scores, a constant that is not constrained is used for denoting an arbitrary object for the intended sort. For example, if we declare a constant x for `Nat` that is the visible sort for natural numbers in a proof score, x can be used to denote an arbitrary natural number. Such constants are constrained with equations, which make it possible to split the state space, or the case. Suppose that the case is split into two: one where x equals 0 and the other where x does not, namely that x is greater than 0. The former is expressed by declaring the following equation:

eq $x = 0$.

The latter is expressed by declaring the following equation:

eq $(x > 0) =$ true .

We are going to mainly describe the proof score of the ith invariant. Let *init* denote any initial state of the system under consideration. All we have to do to show that $pred_i(s, \mathbf{x}_i)$ holds in any initial state is to write the CafeOBJ code, which looks like this:

```
open INV
  red $inv_i(init, \mathbf{x}_i)$ .
close
```

We next write a module, say ISTEP, where two constants s, s' are declared, denoting any state and the successor state after applying a transition rule in the state, and the predicates to prove in each inductive case are expressed as CafeOBJ terms as follows:

```
op $istep_1$ : $\boldsymbol{V}_1$ -> Bool
...
op $istep_n$ : $\boldsymbol{V}_n$ -> Bool
eq $istep_1(\boldsymbol{X})$ = $inv_1(s, \boldsymbol{X}_1)$ implies $inv_1(s', \boldsymbol{X}_1)$ .
...
eq $istep_n(\boldsymbol{X})$ = $inv_n(s, \boldsymbol{X}_n)$ implies $inv_n(s', \boldsymbol{X}_n)$ .
```

These predicates correspond to (3) in the previous section.

In each inductive case, the case is usually split into multiple subcases with basic predicates declared in the CafeOBJ specification. Suppose that we prove that any transition rule denoted by a CafeOBJ action operator a preserves $pred_i(s, \boldsymbol{x}_i)$. As described in the previous section, the case is supposed to be split into the l subcases $case^i_1, \ldots, case^i_l$. Then, the CafeOBJ code showing that the transition rule preserves $pred_i(s, \boldsymbol{x}_i)$ for $case^i_j$ $(j = 1, \ldots, l)$ looks like this:

```
open ISTEP
  Declare constants denoting arbitrary objects.
  Declare equations denoting $case^i_j$.
  Declare equations denoting facts if necessary.
  eq $s'$ = $a(s, \boldsymbol{y})$ .
  red $istep_i(\mathbf{x}_i)$ .
close
```

$\boldsymbol{y}$ is a list of constants that are used as the arguments of the CafeOBJ action operator a, which are declared in this CafeOBJ code and denote arbitrary objects for the intended sorts. In addition to $\boldsymbol{y}$, other constants may be declared in the CafeOBJ code for the case split. Equations are used to express $case^i_j$. If necessary, equations denoting facts about data structures used, etc. may be declared as well. The equation with s' as its left-hand side specifies that s' denotes the successor state after applying any transition rule denoted by a in the state denoted by s.

If $istep_i(\mathbf{x}_i)$ is reduced to `true`, it is shown that the transition rule preserves $pred_i(p, \boldsymbol{x})$ in the subcase j, which corresponds to the proof of the jth formula of (5) in the previous section. Otherwise, we have to strengthen the inductive hypothesis in the way described in the previous section. Let SIH_i be the term denoting SIH_i. Then, instead of $istep_i(\mathbf{x}_i)$, we reduce the following term

$(SIH_i$ `and` $inv_i(s, \mathbf{x}_i))$ `implies` $inv_i(s', \mathbf{x}_i)$,

or

SIH_i `implies` $istep_i(\mathbf{x}_i)$.

5 Example: The NSLPK Authentication Protocol

The NSLPK authentication protocol[10, 11] is the modified version of the NSPK authentication protocol[13] by G. Lowe. The protocol can be described as follows:

$$\begin{array}{l} \text{Msg1 } p \to q : \mathcal{E}_q(n_p, p) \\ \text{Msg2 } q \to p : \mathcal{E}_p(n_p, n_q, q) \\ \text{Msg3 } p \to q : \mathcal{E}_q(n_q) \end{array}$$

Suppose that each principal is given a private/public key pair, and the public counterpart is available to all principals but the private counterpart to its owner only. Given a message m, the one encrypted with the public key given to a principal p is denoted by $\mathcal{E}_p(m)$.

If a principal p wants a principal q to authenticate herself/himself and wants to authenticate q, she/he newly generates a nonce n_p and sends it to q, together with her/his ID, encrypted with q's public key. On receipt of the message, q first decrypts it, obtains a nonce and a principal ID, and checks if the principal ID matches the sender of the message. Then, q newly generates a nonce n_q and sends it to p, together with the received nonce and her/his ID, encrypted with p's public key. On receipt of the message, p first decrypts it and obtains two nonces and a principal ID, and checks if the principal ID equals the sender of the message and one of the nonces is the exact one that p has sent to the sender in this session, which is supposed to convince p that the responder is really q. Then, p sends the other nonce to q, encrypted with q's public key. On receipt of the message, q decrypts it, obtains a nonce and checks if the nonce is the exact one that q has sent to the sender in this session, which supposedly assures q that the initiator is really p.

5.1 Modeling and Description of the Protocol

We suppose that there exist untrustable principals as well as trustable ones. Trustable principals exactly follow the protocol, but untrustable ones may do something against the protocol as well, namely eavesdropping and/or faking messages. The combination and cooperation of untrustable principals is modeled as the most general intruder à la Dolev and Yao[14]. The intruder can do the following:

– Eavesdrop any message flowing in the network.
– Glean any nonce and cipher from the message; however the intruder can decrypt a cipher only if she/he knows the key to decrypt.
– Fake and send messages based on the gleaned information; however the intruder cannot guess unknown nonces.

We first describe the basic data types used to model the protocol. The visible sorts and the corresponding data constructors are as follows:

– `Principal` denotes principals.
– `Random` denotes random numbers, which make nonces unguessable and unique.
– `Nonce` denotes nonces. Given principals p, q and a random number r, $\mathtt{n}(p, q, r)$ denotes a nonce created by p for q. Projections `creator`, `forwhom` and `random` return the first, second and third arguments.
– `Cipher1` denotes ciphers used in Msg1's. Given principals p, q and a nonce n, $\mathtt{enc1}(p, n, q)$ denotes $\mathcal{E}_p(n, q)$. Projections `key`, `nonce` and `principal` return the first, second and third arguments.
– `Cipher2` denotes ciphers used in Msg2's. Given principals p, q and nonces $n1, n2$, $\mathtt{enc2}(p, n1, n2, q)$ denotes $\mathcal{E}_p(n1, n2, q)$. Projections `key`, `nonce1`, `nonce2` and `principal` return the first, second, third and fourth arguments.
– `Cipher3` denotes ciphers used in Msg3's. Given a principal p and a nonce n, $\mathtt{enc3}(p, n)$ denotes $\mathcal{E}_p(n)$. Projections `key` and `nonce` return the first and second arguments.

In addition to those visible sorts, we use the visible sort `Bool` that denotes truth values, declared in the built-in module `BOOL`, where the constants `true` and `false` with usual meanings, and the operators `not_` for negation, `_and_` for conjunction, `_or_` for disjunction and `_implies_` for implication are declared. An underscore `_` indicates where an argument is put.

The three operators (data constructors) to denote the three kinds of messages are declared as follows:

```
op m1 : Principal Principal Principal Cipher1 -> Message
op m2 : Principal Principal Principal Cipher2 -> Message
op m3 : Principal Principal Principal Cipher3 -> Message
```

The visible sort `Message` denotes messages. Projections `creator`, `sender` and `receiver` return the first, second and third arguments of each message. A projection $\mathtt{cipher}i\,(i = 1, 2, 3)$ returns the fourth argument of Msg$i$. A predicate $\mathtt{m}i\mathtt{?}$ checks if a given message is Msg$i$. The first, second and third arguments of each constructor mean the actual creator, the seeming sender and the receiver of the corresponding message. The first argument is meta-information that is only available to the outside observer and the principal that has sent the corresponding message, and that cannot be forged by the intruder, while the remaining arguments may be forged by the intruder.

The network is modeled as a bag (multiset) of messages, which is used as the storage that the intruder can use. The network is also used as each principal's private memory that reminds the principal to send messages, of which the first argument is the principal. Any message that has been sent or put once into the network is supposed to be never deleted from the network because the intruder can replay the message repeatedly, although the intruder cannot forge the first argument. Consequently, the emptiness of the network means that no messages have been sent.

The intruder tries to glean four kinds of quantities from the network. The four kinds of quantities are nonces and three kinds of ciphers. The collections of those quantities gleaned by the intruder are denoted by the following operators:

```
op cnonce : Network -> ColNonce       op cenc1  : Network -> ColCipher1
op cenc2  : Network -> ColCipher2     op cenc3  : Network -> ColCipher3
```

The visible sort `Network` denotes networks and the visible sort `Col`X denotes collections of quantities denoted by the visible sort X. Those operators are defined with equations.

`cnonce` is defined as follows:

```
eq  N \in cnonce(void) = (creator(N) = intruder) .
ceq N \in cnonce(M,NW) = true
    if m1?(M) and key(cipher1(M)) = intruder and nonce(cipher1(M)) = N .
ceq N \in cnonce(M,NW) = true
    if m2?(M) and key(cipher2(M)) = intruder and nonce1(cipher2(M)) = N .
ceq N \in cnonce(M,NW) = true
    if m2?(M) and key(cipher2(M)) = intruder and nonce2(cipher2(M)) = N .
ceq N \in cnonce(M,NW) = true
    if m3?(M) and key(cipher3(M)) = intruder and nonce(cipher3(M)) = N .
ceq N \in cnonce(M,NW) = N \in cnonce(NW)
    if not(m1?(M) and key(cipher1(M)) = intruder and nonce(cipher1(M)) = N) and
       not(m2?(M) and key(cipher2(M)) = intruder and nonce1(cipher2(M)) = N) and
       not(m2?(M) and key(cipher2(M)) = intruder and nonce2(cipher2(M)) = N) and
       not(m3?(M) and key(cipher3(M)) = intruder and nonce(cipher3(M)) = N) .
```

The constant `void` denotes the empty bag and the operator `_,_` denotes the data constructor of nonempty bags. The operator `_\in_` is the membership predicate of bags. The equations say that nonces created by the intruder are always available to the intruder, and a nonce created by one of the other principals is available to the intruder iff there exists a message in the network, and the cipher in the message is encrypted with the intruder's public key and includes the nonce.

`cenc1` is defined as follows:

```
eq  E1 \in cenc1(void) = false .
ceq E1 \in cenc1(M,NW) = true
    if m1?(M) and not(key(cipher1(M)) = intruder) and E1 = cipher1(M) .
ceq E1 \in cenc1(M,NW) = E1 \in cenc1(NW)
    if not(m1?(M) and not(key(cipher1(M)) = intruder) and E1 = cipher1(M)) .
```

The equations say that no ciphers appearing in Msg1's are available if the network is empty, and the intruder glean such a cipher from the network iff there exists a Msg1 in the network and the cipher in the message is not encrypted with the intruder's public key. If the cipher is encrypted with the intruder's public key, the intruder can rebuild the cipher and it is not necessary to collect it. `cenc2` and `cenc3` can be defined likewise.

We are about to describe the OTS modeling the protocol. Two observable values and nine kinds of transition rules are used. The corresponding CafeOBJ observations and actions are as follows:

```
-- observations
bop ur : System -> URand
bop nw : System -> Network
-- actions
bop sdm1  : System Principal Principal Random       -> System
bop sdm2  : System Principal Random Message         -> System
bop sdm3  : System Principal Random Message Message -> System
bop fkm11 : System Principal Principal Cipher1      -> System
bop fkm12 : System Principal Principal Nonce        -> System
bop fkm21 : System Principal Principal Cipher2      -> System
bop fkm22 : System Principal Principal Nonce Nonce  -> System
bop fkm31 : System Principal Principal Cipher3      -> System
bop fkm32 : System Principal Principal Nonce        -> System
```

The hidden sort `System` denotes the state space. The observation `ur` denotes the set of used random numbers and the observation `nw` denotes the network. The first three actions formalize sending messages following to the protocol, and the remaining the intruder's faking messages.

The equations to define `sdm2` are as follows:

```
op c-sdm2 : System Principal Random Message -> Bool
eq c-sdm2(S,Q,R,M)
   = (M \in nw(S) and m1?(M) and receiver(M) = Q and key(cipher1(M)) = Q and
      principal(cipher1(M)) = sender(M) and not(R \in ur(S))) .
--
ceq ur(sdm2(S,Q,R,M)) = R ur(S) if c-sdm2(S,Q,R,M) .
ceq nw(sdm2(S,Q,R,M))
    = m2(Q,Q,sender(M),enc2(sender(M),nonce(cipher1(M)),n(Q,sender(M),R),Q)) , nw(S)
    if c-sdm2(S,Q,R,M) .
ceq sdm2(S,Q,R,M)     = S if not c-sdm2(S,Q,R,M) .
```

The operator `c-sdm2` denotes the effective condition of any transition rule denoted by `sdm2`. `c-sdm2`(s, q, r, m) means that in a state s, there exists a Msg1 m in the network that is addressed to q, the cipher in m is encrypted with q's public key, the principal in the cipher equals the seeming sender, and the nonce generated by q for replying to m is really fresh. If this condition holds, the Msg2 denoted by the term `m2`$(\ldots)$ is put into the network. The juxtaposition operator `__` of '`R ur(S)`' is the data constructor of nonempty sets.

The equations to define `fkm22` are as follows:

```
op c-fkm22 : System Principal Principal Nonce Nonce -> Bool
eq c-fkm22(S,P,Q,N1,N2) = N1 \in cnonce(nw(S)) and N2 \in cnonce(nw(S)) .
--
eq  ur(fkm22(S,P,Q,N1,N2)) = ur(S) .
ceq nw(fkm22(S,P,Q,N1,N2)) = m2(intruder,P,Q,enc2(Q,N1,N2,P)) , nw(S)
    if c-fkm22(S,P,Q,N1,N2) .
ceq fkm22(S,P,Q,N1,N2)     = S if not c-fkm22(S,P,Q,N1,N2) .
```

The equations say that if two nonces are available to the intruder, the intruder can fake and send a Msg2. The constant `intruder` denotes the intruder.

5.2 Proof Scores of Nonce Secrecy

We describe the proof scores showing that nonces are really secret in the protocol, which means that the intruder cannot obtains nonces generated by another principal for yet another one illegally. This can be expressed by the following invariant:

$$\begin{aligned}\textsf{invariant}\ \ &(n\ \texttt{\textbackslash in}\ \texttt{cnonce}(\texttt{nw}(s))\ \texttt{implies}\\ &\quad(\texttt{creator}(n) = \texttt{intruder}\ \texttt{or}\ \texttt{forwhom}(n) = \texttt{intruder}))\,.\end{aligned} \tag{6}$$

It is impossible to prove this invariant alone. We need six more invariants, which are as follows:

$$\textsf{invariant}\ (e1\ \texttt{\\in}\ \texttt{cenc1}(\texttt{nw}(s))\ \texttt{implies}\ \texttt{not}(\texttt{key}(e1) = \texttt{intruder}))\,, \tag{7}$$

$$\textsf{invariant}\ (e2\ \texttt{\\in}\ \texttt{cenc2}(\texttt{nw}(s))\ \texttt{implies}\ \texttt{not}(\texttt{key}(e2) = \texttt{intruder}))\,, \tag{8}$$

$$\textsf{invariant}\ (e3\ \texttt{\\in}\ \texttt{cenc3}(\texttt{nw}(s))\ \texttt{implies}\ \texttt{not}(\texttt{key}(e3) = \texttt{intruder}))\,, \tag{9}$$

$$\begin{aligned}\textsf{invariant}\ \ &(e1\ \texttt{\\in}\ \texttt{cenc1}(\texttt{nw}(s))\ \texttt{and}\ \texttt{principal}(e1) = \texttt{intruder}\\ &\quad \texttt{implies}\ \texttt{nonce}(e1)\ \texttt{\\in}\ \texttt{cnonce}(\texttt{nw}(s)))\,,\end{aligned} \tag{10}$$

$$\begin{aligned}\textsf{invariant}\ \ &(e2\ \texttt{\\in}\ \texttt{cenc2}(\texttt{nw}(s))\ \texttt{and}\ \texttt{principal}(e2) = \texttt{intruder}\\ &\quad \texttt{implies}\ \texttt{nonce}(e2)\ \texttt{\\in}\ \texttt{cnonce}(\texttt{nw}(s)))\,,\end{aligned} \tag{11}$$

$$\textsf{invariant}\ (\texttt{creator}(n) = \texttt{intruder}\ \texttt{implies}\ n\ \texttt{\\in}\ \texttt{cnonce}(\texttt{nw}(s)))\,. \tag{12}$$

The proof of (6) uses (7), (8), (9), (10) and (11) to strengthen inductive hypotheses. The proofs of (10) and (11) use (7), (8), (9) and (12) to strengthen inductive hypotheses. (7), (8), (9) and (12) can be proved independently.

In this paper, we partly show the proof scores showing that any transition rule denoted by sdm2 preserves (6), which uses (10) to strengthen inductive hypotheses. As described in Sect. 4, (6) and (10) are first expressed as CafeOBJ terms, which are denoted by the operators inv1 and inv2 that are declared and defined as follows:

```
op inv1 : System Nonce -> Bool
op inv2 : System Cipher1 -> Bool
eq inv1(S,N) = (N \in cnonce(nw(S))
                implies (creator(N) = intruder or forwhom(N) = intruder)) .
eq inv2(S,E1) = (E1 \in cenc1(nw(S)) and principal(E1) = intruder
                 implies nonce(E1) \in cnonce(nw(S))) .
```

A constant n for Nonce and a constant e1 for Cipher1 are also declared. The predicates to prove in each inductive case are next expressed as CafeOBJ terms, which are denoted by the operators istep1 and istep2 that are declared and defined as follows:

```
op istep1 : Nonce -> Bool
op istep2 : Cipher1 -> Bool
eq istep1(N) = inv1(s,N) implies inv1(s',N) .
eq istep2(E1) = inv2(s,E1) implies inv2(s',E1) .
```

The constants s, s' for System are also declared.

In the inductive case under consideration, the case is split into six subcases based on the following predicates:

$$\begin{aligned}
\mathrm{bp1} &\overset{\mathrm{def}}{=} \texttt{c-sdm2(s,p10,r10,m10)}\\
\mathrm{bp2} &\overset{\mathrm{def}}{=} \texttt{(sender(m10) = intruder)}\\
\mathrm{bp3} &\overset{\mathrm{def}}{=} \texttt{(n(p10,intruder,r10) = n)}\\
\mathrm{bp4} &\overset{\mathrm{def}}{=} \texttt{(nonce(cipher1(m10)) = n)}\\
\mathrm{bp5} &\overset{\mathrm{def}}{=} \texttt{(p10 = intruder)}
\end{aligned}$$

The constants p10 for Principal, r10 for Random and m10 for Message are used as the arguments of c-sdm2. Then, the case is split as follows:

1	bp1	¬bp2			
2		bp2	bp3		
3			¬bp3	¬bp4	
4				bp4	bp5
5					¬bp5
6	¬bp1				

Each case is denoted by the predicate obtained by connecting ones appearing in the row with conjunction.

The proof score for subcase 5 is shown.

```
open ISTEP
-- arbitrary objects
  op p10 : -> Principal .  op r10 : -> Random .
  op m10 : -> Message .    op nw10 : -> Network .
-- assumptions
  -- eq c-sdm2(s,p10,r10,m10) = true .
  eq nw(s) = m10 , nw10 .                     eq m1?(m10) = true .
  eq receiver(m10) = p10 .                    eq key(cipher1(m10)) = p10 .
  eq principal(cipher1(m10)) = sender(m10) .  eq r10 \in ur(s) = false .
  --
  eq sender(m10) = intruder .
  eq (n(p10,intruder,r10) = n) = false .
  eq nonce(cipher1(m10)) = n .
  eq (p10 = intruder) = false .
-- successor state
  eq s' = sdm2(s,p10,r10,m10) .
-- check if the predicate is true.
  red inv2(s,cipher1(m10)) implies istep1(n) .
close
```

The condition that there exists a message denoted by `m10` in the network denoted by `nw(s)` is expressed by the equation "`eq nw(s) = m10 , nw10 .`" Except for the conjunct corresponding to this condition, each conjunct in bp1 is expressed by one equation.

For the remaining five subcases, similar proof scores can be written, which do not use any other invariant to strengthen inductive hypotheses. For subcase 6 where the effective condition is false, it is not necessary to write the proof score in theory because nothing changes. But, from an engineering point of view, the proof score is worth writing because the specification may be miswritten.

6 Discussion

If we write proof scores of invariants in CafeOBJ, the compositional proofs of invariants described in Sect. 3 has the following advantages:

1. CafeOBJ reduces a logical formula into an exclusive-or normal form à la Hsiang[15], of which response time crucially depends on the length of the formula, essentially the possible number of `or`'s in it. The compositional proofs of invariants make it possible to focus on each conjunct $pred_i(s, \mathbf{x}_i)$ $(i = 1, \ldots, n)$ of a large formula $pred(s, \mathbf{x}_1, \ldots, \mathbf{x}_n)$ and relieve the complexity of the reduction.
2. Since we prove each conjunct of a large formula individually, case analyses can be done w.r.t. this conjunct only. This can ease the complexity of case analyses. Suppose that we have to consider N_i subcases for $pred_i(s, \mathbf{x}_i)$ and N_j subcases for $pred_j(s, \mathbf{x}_j)$ in an inductive case. If we try to prove $pred_i(s, \mathbf{x}_i) \wedge pred_j(s, \mathbf{x}_j)$ together, then we may have to consider $N_i \times N_j$ subcases in the inductive case.

It is often the case that a large formula is divided into smaller ones to ease the proof. In our method, however, any two of such smaller formulas may depend on each other in the sense that the proof of one uses the other to strengthen inductive hypotheses in inductive cases and vice versa. Existing proof assistants such as Isabelle/HOL[16] may not allow to divide a formula into such two subformulas.

It is significant to split the case into multiple subcases in an inductive case and find another inductive hypothesis (or a lemma) to strengthen the direct one of what being proved in order to make progress on a proof. The former can be done based on predicates used in a specification such as `_=_` and `_\in_` in the NSLPK protocol. If you encounter a subcase where the formula stating a property under discussion is reduced to `false`, the subcase may be unreachable and you may conjecture another invariant.

Although the OTS/CafeOBJ method is not specific to object-orientation, it can be easily applied to object-based distributed systems by modeling an object as an OTS. In the behavioral specification in CafeOBJ, modeling and verification techniques that are specific to object-orientation have been studied[17], which can be incorporated in the OTS/CafeOBJ method.

In addition to the proof that nonces are really secret in the NSLPK protocol, we have proved that the protocol has one-to-many agreement property[18], which is expressed as the following two invariants:

$$\begin{array}{ll}
\mathsf{invariant} & (\mathtt{not}(p = \mathtt{intruder})\ \mathtt{and} \\
& \mathtt{m1}(p, p, q, \mathtt{enc1}(q, \mathtt{n}(p, q, r), p))\ \backslash\mathtt{in}\ \mathtt{nw}(s)\ \mathtt{and} \\
& \mathtt{m2}(q1, q, p, \mathtt{enc2}(p, \mathtt{n}(p, q, r), n, q))\ \backslash\mathtt{in}\ \mathtt{nw}(s) \\
& \mathtt{implies}\ \mathtt{m2}(q, q, p, \mathtt{enc2}(p, \mathtt{n}(p, q, r), n, q))\ \backslash\mathtt{in}\ \mathtt{nw}(s))\,, \\
\mathsf{invariant} & (\mathtt{not}(q = \mathtt{intruder})\ \mathtt{and} \\
& \mathtt{m2}(q, q, p, \mathtt{enc2}(p, n, \mathtt{n}(q, p, r), q))\ \backslash\mathtt{in}\ \mathtt{nw}(s)\ \mathtt{and} \\
& \mathtt{m3}(p1, p, q, \mathtt{enc3}(q, \mathtt{n}(q, p, r)))\ \backslash\mathtt{in}\ \mathtt{nw}(s) \\
& \mathtt{implies}\ \mathtt{m3}(p, p, q, \mathtt{enc3}(q, \mathtt{n}(q, p, r)))\ \backslash\mathtt{in}\ \mathtt{nw}(s))\,.
\end{array}$$

The former describes that if a principal p has sent a Msg1 to a principal q and has received a Msg2 in response to the Msg1 seemingly from q, although the actual sender $q1$ might be the intruder, then the Msg2 originates from q, and the latter describes a similar phenomenon. We need (6) and eight more invariants to prove these two invariants. The proof has been done by writing proof scores in CafeOBJ likewise.

The modeling and proof scores described in [19] are different than those described in this paper.

References

1. Gries, D., Schneider, F.B.: A Logical Approach to Discrete Math. Texts and Monographs in Computer Science. Springer, NY (1993)
2. Goguen, J., Malcolm, G., eds.: Software Engineering with OBJ: Algebraic Specification in Action. Kluwer Academic Publishers (2000)

3. Ogata, K., Futatsugi, K.: Formally modeling and verifying Ricart&Agrawala distributed mutual exclusion algorithm. In: APAQS '01, IEEE CS Press (2001) 357–366
4. Ogata, K., Futatsugi, K.: Formal analysis of Suzuki&Kasami distributed mutual exclusion algorithm. In: FMOODS '02, Kluwer Academic Publishers (2002) 181–195
5. Ogata, K., Futatsugi, K.: Formal analysis of the *i*KP electronic payment protocols. In: ISSS 2002. Volume 2609 of LNCS., Springer (2003) 441–460
6. Ogata, K., Futatsugi, K.: Formal verification of the Horn-Preneel micropayment protocol. In: VMCAI 2003. Volume 2575 of LNCS., Springer (2003) 238–252
7. CafeOBJ: CafeOBJ web page. `http://www.ldl.jaist.ac.jp/cafeobj/` (2001)
8. Diaconescu, R., Futatsugi, K.: CafeOBJ report. AMAST Series in Computing, 6. World Scientific, Singapore (1998)
9. Diaconescu, R., Futatsugi, K.: Behavioural coherence in object-oriented algebraic specification. Journal of Universal Computer Science **6** (2000) 74–96
10. Lowe, G.: An attack on the Needham-Schroeder public-key authentication protocol. Inf. Process. Lett. **56** (1995) 131–133
11. Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In: TACAS '96. LNCS 1055, Springer (1996) 147–166
12. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading, MA (1988)
13. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Comm. ACM **21** (1978) 993–999
14. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE Trans. Inform. Theory **IT-29** (1983) 198–208
15. Hsiang, J.: Refutational Theorem Proving using Term Rewriting Systems. PhD thesis, University of Illinois at Champaign-Urbana (1981)
16. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL – A Proof Assistant for Higer-Order Logic. LNCS 2283. Springer (2002)
17. Diaconescu, R., Futatsugi, K., Iida, S.: Component-based algebraic specification and verification in CafeOBJ. In: World Congress on Formal Methods. Volume 1709 of LNCS., Springer (1999) 1644–1663
18. Lowe, G.: A hierarchy of authentication specifications. In: 10th IEEE Computer Security Foundations Workshop, IEEE CS Press (1997) 31–43
19. Ogata, K., Futatsugi, K.: Rewriting-based verification of authentication protocols. In: WRLA '02. Volume 71 of ENTCS., Elsevier Science Publishers (2002)

Managing the Evolution of .NET Programs

Susan Eisenbach[1], Vladimir Jurisic[1], and Chris Sadler[2]

[1] Department of Computing
Imperial College
London, UK SW7 2BZ
{sue,vj98}@doc.ic.ac.uk
[2] School of Computing Science
Middlesex University
London, UK N14 4YZ
c.sadler@mdx.ac.uk

Abstract. The component-based model of code execution imposes some requirements on the software components themselves, and at the same time lays some constraints on the modern run-time environment. Software components need to store descriptive metadata, and the run-time system must access this 'reflectively' in order to implement *dynamic linking*. Software components also undergo *dynamic evolution* whereby a client component experiences the effects of modifications, made to a service component even though these occurred after the client was built.

We wanted to see whether the dynamic linking mechanism implemented in Microsoft's .NET environment could be utilized to maintain multiple versions of components. A formal model was developed to assist in understanding the .NET mechanism and in describing our way of dealing with multiple versions. This showed that .NET incorporates all the features necessary to implement such a scheme and we constructed a tool to do so.

1 Introduction

The dynamic link library (DLL) was invented to allow applications running on a single system to share code. Because it is linked at run-time, when a DLL is corrected or enhanced, the effect on the client applications is experienced immediately.

This dynamic evolution is a benefit provided that two conditions are met. Firstly, successive versions of the DLL must maintain backward compatibility. Secondly, on any particular system, any given version of a DLL can only be replaced by a later version. Failure to meet the first condition results in the *upgrade* problem whilst failure to meet the second results in the *downgrade* problem. Together these problems contribute to "DLL hell" which has been well described in [21, 20]. Many Microsoft support personnel report [2] DLL hell as the single most significant user problem that they are called upon to respond to.

There is another way to solve these problems, and that is by allowing the system to keep multiple versions of the same DLL such that each application is linked to a compatible version. Although it reduces the amount of code-sharing and loses the benefits of dynamic evolution, this idea is the one that is implemented in Microsoft's .NET environment. .NET aims to provide developers with a component-based execution model. Any application will consist of a suite of co-operating software *components* amongst whose members control is passed during execution. The CLR (Common Language Runtime) needs to be 'language-neutral' which means it should be possible to write a component in any one of a variety of languages and have it executed by the CLR; and also that it should be possible to pass data between components written in different languages (using the Common Type System – CTS).

The usual way to do this is with an intermediate language which the high-level languages compile to, and to constrain their data types to those recognized by the intermediate language. .NET achieves this with IL (Intermediary Language). Whereas most systems with an intermediate language have run-time systems that interpret the intermediate code (usually for portability reasons), the CLR uses a 'just-in-time' (JIT) compiler to translate IL fragments (primarily method bodies) into native code at run-time.

The native code must run within some previously loaded context, and when the JIT compiler encounters a reference external to the context, it must invoke some process to locate and establish the appropriate context before it can resolve the reference. A reference to a not yet loaded entity (type or type member) within the current, or another (previously loaded) component, causes a classloader to load the corresponding type definition. When the reference is to an entity external to all loaded components, it will be necessary to load the new component (or .NET *assembly*) via the Library Loader.

The design of the .NET assembly reveals some details about how this is accomplished, with each assembly carrying versioning information structured into four fields – Major, Minor, Revision and Build. This allows many versions of the same DLL to co-exist in the system. To make this system useful two things are required. Firstly, we have to agree on the semantics of what constitutes Major, Minor *etc.* Secondly, we have to be able to exercise some control over which version will be loaded by the Library Loader when a new component is needed at run-time. The Fusion utility in .NET is configured to find, according to some *policy*, the appropriate component and to pass its pathname to the Library Loader.

Each assembly normally contains at least one code module[1]. If we restrict it to exactly one module, then Microsoft's definition of an assembly coincides with nearly everybody else's definition of a component, so we shall adopt that restriction. It is the module that is the basic unit for loading in the .NET run-time. Each module contains a piece of code (*e.g.* a class definition) translated into an IL binary. In addition, the module incorporates metadata that records the name and location of every type and member defined within the module. This metadata is referenced by the CLR whenever it needs to instantiate an object, access an attribute, invoke a method or request a type from the class loader.

In addition to the above (module) features, the assembly contains its own metadata, consisting of *type* metadata and a *manifest*. These features distinguish .NET from other current run-time environments. The type metadata records external type and member references indexed against a manifest entry. The manifest contains details (name, version number, public key, *etc.*) of each external component needed by the assembly. This information is what permits dynamic linking between the executing components and, in .NET it must be *explicitly* provided by the programmer at compile-time. By contrast,

[1] An assembly can actually consist of resources only, but this is not common.

the Java Virtual Machine has a simpler dynamic linking mechanism that *implicitly* resolves references along various ‘classpaths’. The significance of this difference is that the JVM can only accommodate one version of each component at a time – normally, the first one encountered in the classpath unless elaborate mechanisms are employed to invoke custom classloaders [27]. By forcing explicit references .NET allows the programmer to bind the client code to particular versions of the referenced services. At run-time, those exact services will be accessed, provided that they still exist and that the manifest binding is not superseded by an alternative policy.

The assembly metadata is the key to the management of both dynamic linking and the sensible evolution of components. Some of this information can be accessed at run-time by developers via a Managed Reflection API, but it cannot be manipulated by this means. Instead, a set of ‘unmanaged’ COM interfaces allows full read/write access via C++ hacking. By this means we can see a possible way out of DLL Hell: firstly, when a component evolves, we do not need to replace the old version with the new, because .NET lets us distinguish between them by means of version numbers. Secondly, by manipulating the manifests of client components we can ensure that suitable clients can benefit from the post-compile-time evolution of their service components, thus fulfilling the main benefit of evolving DLLs. By the same means we can leave other clients’ bindings undisturbed and thus overcome DLL Hell. To do this, we need to –

1. establish the characteristics of a set of inter-related components – called, in .NET, the Global Assembly Cache (GAC);
2. investigate how these inter-relationships may be affected by component evolution, and identify any requirements this may impose;
3. reflect the requirements in new policies.

Section two tackles points (1) and (2) via a formal model. Section three describes a tool Dejavue.NET [17] based on the model which attempts to implement (3). In section four we report on other recent work which has addressed this problem, and in section five we give some indications of where this work might lead in the future.

2 Modelling the Component Cache

To investigate the characteristics of systems of components with dynamic evolution we modelled the system in Alloy [16]. Alloy is a notation that supports the declarative description of systems that have relational structures. Alloy is a first-order relational calculus with transitive closure. Alloy is supported by Alcoa, the Alloy analyser [8], which allows us to analyse our Alloy model to check for consistency and to generate example situations which we may not have considered.

The Alcoa tool provides a visualiser which will display example structures graphically. This representation is easy to interpret. We can see how the components have been joined together to form a system. The figures in this paper were generated by this visualisation tool. The visualisation tool is quite flexible, allowing us to omit parts of the model and to show fields either within an object or with an arrow from the object. In figure 1 we use both ways to show the components of a system.

To work out the characteristics needed to avoid dynamic linking problems caused by having multiple versions of components we modelled the system looking for undesirable behaviour. Several properties not in the original system needed to be included to simulate the behaviour we desired. The complete Alloy model is available at [12].

The type $\mathcal{S}$ of services is atomic[2]. A service $s : \mathcal{S}$, is either a programming language type (such as a class) or a member (field or method) represented by the underlying .NET Common Type System. Components $c : \mathcal{C}$ consist of a name, a version, an optional `main` method, and three sets. These are the set of services that the component imports – import, the set of services that the component exports – export, and a set of components that provide the services that are listed in the imports – req. These sets model the metadata. In a programming context, the services that a component imports are the references that need to be resolved so that the component can be linked.

Definition 1 (Component). *A component $c : \mathcal{C}$ is defined as:*

$$< name : id, \mathsf{version} : ord, \mathsf{main} : optional,$$
$$\mathsf{import} : \mathcal{P}(\mathcal{S}), \mathsf{export} : \mathcal{P}(\mathcal{S}), \mathsf{req} : \mathcal{P}(\mathcal{C}) >$$

We can extract an element of a component by its name, *e.g.* $c.\mathsf{main}$. We assume the ordering of version is temporal, that is, if two components have the same name but different version numbers, the one with the greater version number was produced more recently. The four number Microsoft version numbers are totally ordered so a single number is sufficient. The first of our version numbers is denoted by v_0 or `0`. Given a version number v, the one that follows is $next(v)$.

In our model services (class, method or field) don't exist in isolation, they only exist if they are exported from some component. The predicate in property 1 states this.

Property 1 (ServiceInComponent).

$$\forall s \in \mathcal{S}, \exists c \in \mathcal{C}\ (s \in c.\mathsf{export})$$

In our model two components with the same name and version are considered the same component, whatever their other elements are, whereas two components with the same name but different version are distinguishable. This is stated in property 2.

Property 2 (Unique).

$$\forall c_1, c_2 \in \mathcal{C}\ ((c_1.\mathsf{name} = c_2.\mathsf{name} \wedge c_1.\mathsf{version} = c_2.\mathsf{version}) \Rightarrow c_1 = c_2)$$

For a given component, there is a relationship between the set of services import that are imported by a component and the set of components req. Namely, the set req only contains components that export the required services. More than one component may export a given service so a given component's req set is not necessarily unique.

Property 3 (ExportsFoundInRequiredComponents).

$$\forall c \in \mathcal{C}, \forall s \in c.\mathsf{import}, \exists c_1 \in c.\mathsf{req}\ (s \in c_1.\mathsf{export})$$
$$\forall c_1, c_2 \in \mathcal{C}\ (c_2 \in c_1.\mathsf{req} \Rightarrow c_1.\mathsf{import} \cap c_2.\mathsf{export} \neq \varnothing)$$

Two different components with the same name (and hence different version numbers) may cause problems if one tries to link to both of them. So it is not possible for one component to import services from two components with the same name[3]. Nor is

[2] For a declared type $\mathcal{T}$, $t \in \mathcal{T}$ and $t : \mathcal{T}$ will be used interchangeably.

[3] This restriction does not exist in .NET although it does exist in many current .NET language implementations [18].

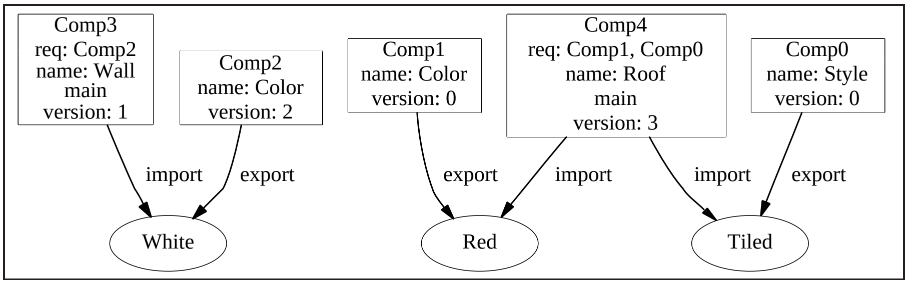


Fig. 1. A Closed set of five components containing two versions of one component.

it possible in the transitive closure of the req sets of a given component to require two components with the same name.

*Property 4 (*OnlyOneVersion*).*

$$\forall c, c_1, c_2 \in \mathcal{C}\ ((c_1 \neq c_2 \wedge c_1 \in c.\mathsf{req} \wedge c_2 \in c.\mathsf{req}) \Rightarrow c_1.\mathsf{name} \neq c_2.\mathsf{name})$$
$$\forall c, c_1 \in \mathcal{C}\ (c_1 \in c.^{*}\mathsf{req} \Rightarrow c.\mathsf{name} \neq c_1.\mathsf{name})$$

Figure 1, automatically generated from the Alloy model at [12][4], contains an example of a set of components that have already evolved over time. `Comp3` and `Comp4` have main methods and the other three components do not. There are two components named `Color`. The earlier version `Comp1` exports a service `Color.Red`. The later version of this component `Comp2` exports `Color.White`. `Comp4` requires services from `Comp0` and `Comp1` as can be seen from its req set whereas `Comp0` is not dependent on any other components.

It is possible that a service required by one of a set of components is not available. We define the predicate Closed to test for whether all services required by the components within a set are available in that set.

Definition 2 (Closed)**.**

$$\mathsf{Closed} \subseteq \mathcal{P}(\mathcal{C})$$
$$\mathsf{Closed}(C) \Leftrightarrow \forall c \in C, \forall s \in c.\mathsf{import}, \exists c_i \in C\ (s \in c_i.\mathsf{export})$$

We need to model the component cache G. By requiring G to be Closed it will have the property that services required from any component within the set are always available within the set itself.

Definition 3 (GlobalComponentCache)**.** *A global component cache $G : \mathcal{G}$ is a finite set of components s.t.* Closed(G).

Next we define a program $P : \mathcal{P}$. Programs consist of a set of components, such that one c contains a main method and there are no unresolved references, in any of the components. This component c is the starting point (main holds for this component)

[4] Alloy generated figures have had a small amount of hand editing of the labels labels to make the models easier to understand.

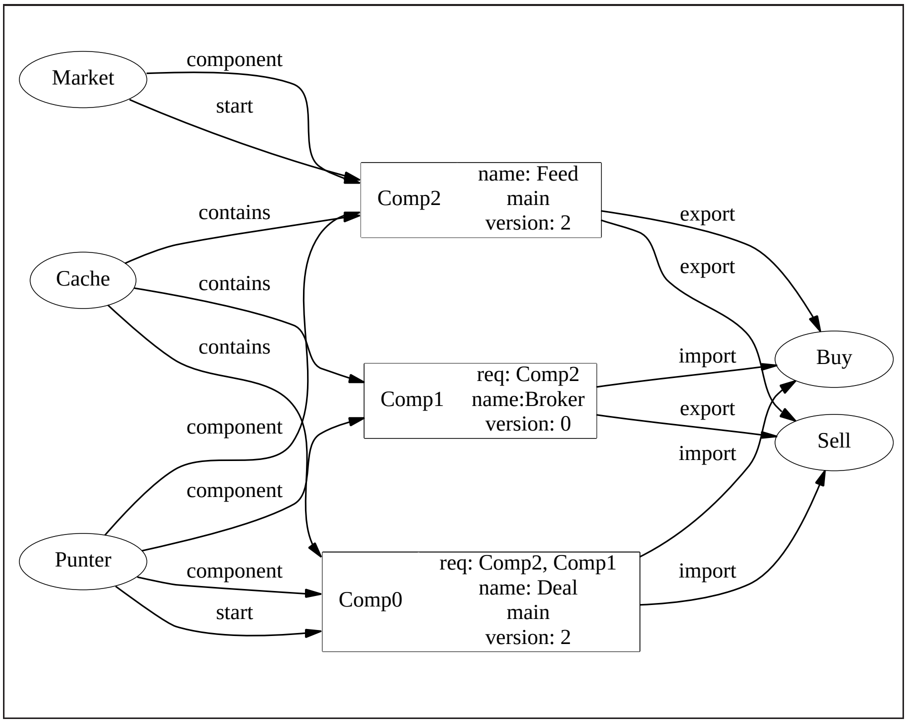


Fig. 2. A cache containing two programs.

and all other components are those that can be reached from this component by the transitive closure of c.req. Figure 2, automatically generated from the Alloy model, shows a cache G containing three components and two programs.

Definition 4 (program)**.** *A program $P : \mathcal{P}$ is a finite set of components including a component c with a* main *s.t.*

$$P = \{c\} \cup c.\mathsf{req}^*$$

Programs don't exist in isolation. They are held within caches.

*Property 5 (*ProgramInCache*).*

$$\forall P : \mathcal{P}, \exists G : \mathcal{G}\ (P \subseteq G)$$

Sometimes we need to know whether a set of components (*e.g.* a program or a cache) uses the latest versions of components that provide needed services available in that set[5]. This we call WellVersioned.

[5] This model assumes that there are no side-effects – everything provided by a component is in its exports.

Definition 5 (WellVersioned).

$$\begin{array}{l}\mathsf{WellVersioned} \subseteq \mathcal{P}(\mathcal{C})\\ \mathsf{WellVersioned}(C) = \forall c \in C, \forall c_1 \in C \setminus \{c\}\ ((c.\mathsf{name} = c_1.\mathsf{name}) \Rightarrow\\ \qquad\qquad ((c.\mathsf{version} \geq c_1.\mathsf{version}) \vee \neg\mathsf{Closed}(c.\mathsf{req}^* \cup \{c\} \setminus \{c_1\})))\end{array}$$

A component can be added to a set of components (*e.g.* a program or a cache) only if adding it doesn't lead to unresolved references in the augmented set. Before adding a component a unique version number has to be given to the component. If the component to be added is the first possessing its name then its version number is 0. If another component already exists within the set with the same name, then the version number of the new component needs to be greater than all other version numbers of components with the same name.

Definition 6 (NewNum).

$$\begin{array}{lll}\mathsf{NewNum} : (\mathcal{P}(\mathcal{C}), \mathcal{C}) \longrightarrow \mathcal{C} & & \\ \mathsf{NewNum}(C, c) & = & < n, 0, m, i, e, r >,\\ & & \textbf{if } \nexists c_1 \in C\ (c_1.\mathsf{name} = c.\mathsf{name})\\ & = & < n, next(c_1.\mathsf{version}), m, i, e, r >,\\ & & \textbf{if } (\exists c_1 \in C\ (c_1.\mathsf{name} = c.\mathsf{name}) \wedge\\ & & \forall c_2 \in C\ (c_2.\mathsf{name} = c.\mathsf{name} \Rightarrow c_2.\mathsf{version} \leq c_1 \mathsf{version}))\\ & where\ c =< n, v, m, i, e, r > & \end{array}$$

Definition 7 (Add).

$$\begin{array}{ll}\mathsf{Add} : (\mathcal{P}(\mathcal{C}), \mathcal{C}) \longrightarrow \mathcal{P}(\mathcal{C}) & \\ \mathsf{Add}(C, c) = C \cup \{\mathsf{NewNum}(C, c)\}, & \textbf{if } \mathsf{Closed}(C \cup \{\mathsf{NewNum}(C, c)\})\\ \qquad\quad\ = C, & otherwise\end{array}$$

As we add components to a cache (or a program) some housekeeping needs to be done, to ensure that the latest versions of components that don't break code will be dynamically linked. In particular before a given component starts executing we have to update the pointers that indicate which components will get chosen to resolve references. These are modelled as the req set for each component. None of our definitions so far, alter this set for a given component, even when newer versions of components have been added. Firstly we need to find out which is the latest version of a component that provides whatever was required. There will always be such a component since there is no operation that can remove the Closed property from a set.

Definition 8 (Latest).

$$\begin{array}{lll}\mathsf{Latest} : (\mathcal{P}(\mathcal{C}), \mathcal{C}) \longrightarrow \mathcal{C} & & \\ \mathsf{Latest}(C, c) = c_1, & \textbf{if} & c \in C \wedge c_1 \in C \wedge\\ & & c.\mathsf{name} = c_1.\mathsf{name} \wedge c.\mathsf{export} \subseteq c_1.\mathsf{export} \wedge\\ & & \forall c_2 \in C\ (c.\mathsf{name} = c_2.\mathsf{name} \Rightarrow\\ & & (c_2.\mathsf{version} \leq c_1.\mathsf{version} \vee c.\mathsf{export} \subsetneq c_2.\mathsf{export}))\\ \qquad\qquad = undef,\ otherwise & & \end{array}$$

Once we know which version of a component should be linked we need to be able to create for each component a new req set. This we do with Provides which takes a component's req set within a cache G and returns a set with the newest components that contain the required exports.

Definition 9 (Provides)**.**

$$\begin{aligned}&\mathsf{Provides} : (\mathcal{P}(\mathcal{C}), \mathcal{P}(\mathcal{G})) \longrightarrow \mathcal{P}(\mathcal{C})\\ &\mathsf{Provides}(R, G) = \{c_i \mid \exists c \in R\ (c_i = \mathsf{Latest}(G, c) \wedge R \subseteq G \wedge c_i \in G)\}\end{aligned}$$

We now have the functionality needed to reconfigure all the pointers so that programs can evolve. We use the $\oplus$ symbol to override the values of a component's req set.

Definition 10 (Reconfigure)**.**

$$\begin{aligned}&\mathsf{Reconfigure} : \mathcal{P}(\mathcal{C}) \longrightarrow \mathcal{P}(\mathcal{C})\\ &\mathsf{Reconfigure}(G) = \{c_i \mid \exists c \in G\ (c_i = c \oplus \mathsf{Provides}(c.\mathsf{req}, G))\}\end{aligned}$$

Finally we need an algorithm for evolving the global cache. We only put into the cache (using Add) new components whose references can be completely satisfied by the components already in the cache. After we put a component in we need to do some housekeeping. Firstly, all components in the global cache (including the new one) have to have their req sets updated, so they now get their import services from the latest versions of components that provide them. Secondly, any components which are no longer needed should be removed.

Definition 11 (Evolve)**.**

$$\begin{aligned}&\mathsf{Evolve} : (\mathcal{P}(\mathcal{C}), \mathcal{C}) \longrightarrow \mathcal{P}(\mathcal{C})\\ &\mathsf{Evolve}(G, c) = \mathsf{Reconfigure}(\mathsf{Keep}(\mathsf{Add}(G, c), c))\\ &\quad where\\ &\qquad \mathsf{Keep}(C, c) = \{c_i \mid c_i \in C \wedge\\ &\qquad\qquad (c.\mathsf{name} \neq c_i.\mathsf{name}) \vee\\ &\qquad\qquad ((c.\mathsf{name} = c_i.\mathsf{name}) \wedge ((c.\mathsf{version} < c_i.\mathsf{version}) \vee\\ &\qquad\qquad (c_i.\mathsf{version} \leq c.\mathsf{version} \wedge c_i.\mathsf{export} \subsetneq c.\mathsf{export})))\}\end{aligned}$$

Theorem 1. *If a set of components C is* Closed *and* WellVersioned *then*

$$\forall c : \mathcal{C}\ (\mathsf{Evolve}(C, c))\ \textit{is}\ \mathsf{Closed}\ \textit{and}\ \mathsf{WellVersioned}\ .$$

Proof. For each of the properties, by contradiction.

Unfortunately Alloy cannot be used to check the theorem (using its `assert`) since it is higher order. All one can see is that the theorem is not inconsistent with the model. This can be seen by adding the conditions Closed and WellVersioned of the input set and the output set to Evolve. The models generated are the same as those without these conditions.

The sets of components we are actually interested in are caches.

Corollary 1. *If the set of components in a* GlobalComponentCache G *is* WellVersioned *then*

$$\forall c : \mathcal{C}\ (\mathsf{Evolve}(G, c))\ \textit{is}\ \mathsf{Closed}\ \textit{and}\ \mathsf{WellVersioned}\ .$$

Using Alloy to build and test the model helped refine it considerably. Running early versions immediately threw up undesirable behaviour. Most of the properties (such as ExportsFoundInRequiredComponents or those in Component) were added to stop the converse from being possible. We convinced ourselves that the definition of Evolve behaved as we wanted by using Alloy to show that there were no models that failed to meet our theorem.

This model was built using features that are all available from .NET assemblies. New programs are WellVersioned and the changes during cache evolution are designed to maintain WellVersionedness. If the component cache is evolved according to our model, then the programs built from the cache components should contain the latest versions of components that do not cause problems.

Of course there are other ways of modelling a component cache. The formalization cannot cope with components that engage in cyclic import relationships because we only add one component at a time whilst maintaining Closed at all times. This restriction reflects current .NET development tools (but not .NET itself). Also in our model the set of services that are imported and the set of components that provide these services are not explicitly paired. Nor is there an explicit list of connections between components. The model might have been in some sense more natural, if these were included, but then it would have deviated from what actually occurs in .NET metadata.

We set out to build a versioning tool, Dejavue.NET, that could follow the precepts of our model to manage evolution in .NET. The prototype is described in the next section.

3 Dejavue.NET

Before designing a tool based on our model, a number of practical problems needed to be solved. Our model envisages a single version number with the convention that a *higher* number indicates a *later* version. Early Beta versions of .NET revealed a versioning policy that would resolve references against an assembly with higher Revision and Build numbers than the compile-time version, provided the Major and Minor numbers were the same. No checking was done to determine whether or not the two versions were binary compatible. Presumably, this approach did not solve the problems for the Beta testers because in later Beta releases and indeed in the .NET official release, this policy was dropped in favour of a default policy of an exact match between the compile-time and the link-time versions – thus ruling out dynamic evolution altogether[6].

Until there is some agreement amongst developers about what sort of evolution will cause change in which parts of the version number, it will never be possible to implement sufficiently sensitive policies based on version numbers alone. For the tool therefore, we will stick to the idea that a higher number indicates a later version, and

[6] So the approach is that rolling forward is only safe if explicit policy statements have been made.

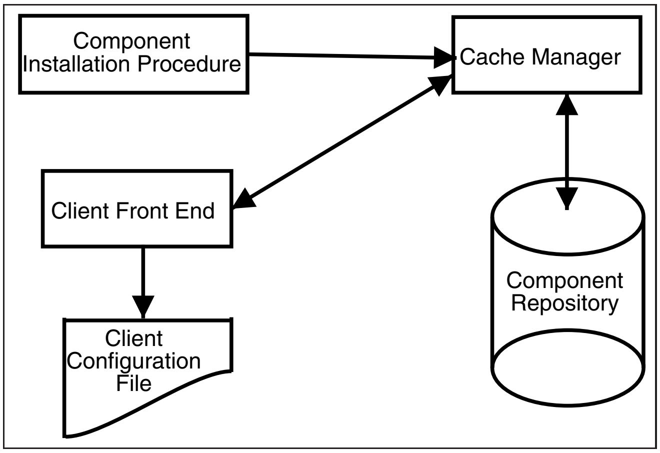


Fig. 3. High level architecture of the single-machine Dejavue.NET.

we will use Closed and WellVersioned properties to determine which later version, if any, maintains compatibility.

Initially we planned to manipulate assembly metadata in situ in order to keep client components abreast of service component evolution. This idea was frustrated in two ways – firstly, .NET encrypts assemblies in the GAC and this puts the metadata out of reach of unmanaged API manipulations. Secondly, in the .NET release the GAC assemblies were placed out of reach of even the managed API. This is another blow (like the inflexible versioning policy) to dynamic evolution.

However, before it looks in the GAC, the Fusion utility consults a sequence of configuration files. Whoever can control these files has the capability to override, with more up-to-date possibilities, the assembly metadata relationships established at compile-time. This provides a mechanism for implementing a more evolutionary dynamic linking policy, utilising the operations and predicates identified within our theoretical model.

Instead of using the GAC, our tool mimics the GAC with its own Component Repository (CR) which is maintained by a CacheManager module (see Figure 3). This faithfully implements the installation regime defined in our model via a Component Installation Routine. A second function in the CacheManager, the Redirection Information Retrieval Routine, traverses the CR dependency tree to construct, for a given component, a configuration file containing the current (link-time) dependency information.

The final part of the tool is the ClientFrontEnd. This allows users to execute the most up-to-date version of an executable component c, in the following steps:

1. requests the CacheManager to interrogate the CR to produce a list of executable components;
2. allows the user to select the required component c;
3. passes the component name back to the CacheManager to construct ‘on the fly’ the appropriate configuration file, which is stored in the ClientFrontEnd’s working directory;
4. invokes c.

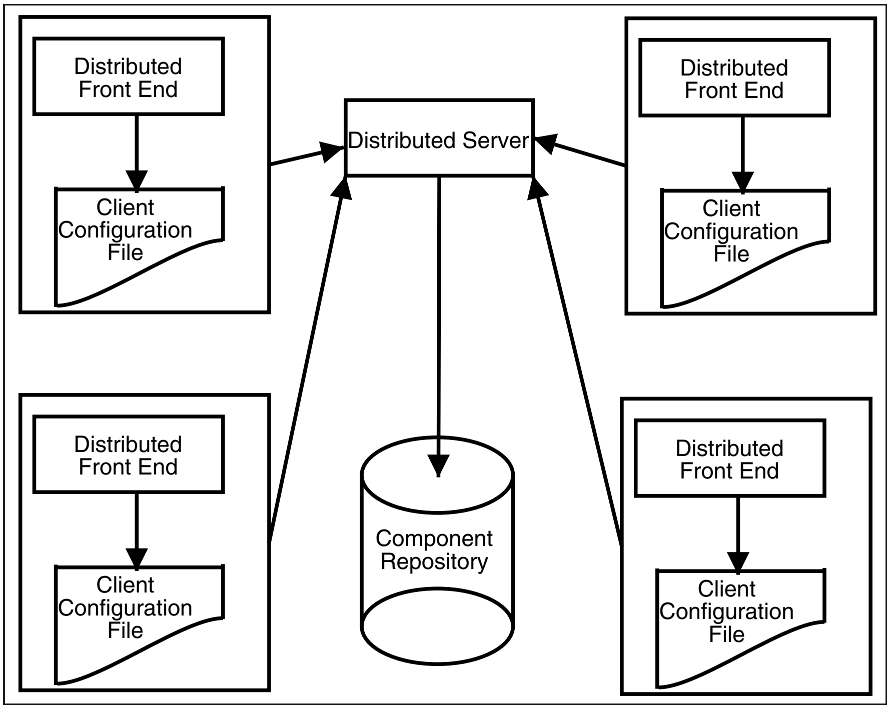


Fig. 4. Distributed Dejavue.NET architecture.

Once this has been done, any references external to *c* will be resolved by invoking the Fusion utility. Instead of looking in the GAC however, Fusion will use the configuration file to locate components in the CR.

A tool of this sort can help an end-user to keep up-to-date provided all updates are downloaded onto the local system and introduced to the Component Repository via the Cache Manager. This is not a very practical solution, especially from the point of view of component developers. Instead, a prototype distributed version of the tool has been developed (see Figure 4). In this scheme the Component Repository is hosted on a server which performs the functions of the Cache Manager. Each client then has a local copy of the Client Front End which maintains local copies of the application configuration files.

This scheme could slow down execution times if all external references have to be resolved across the network, so efficiency considerations may dictate some local caching mechanism. In addition, where support is required for distributed component developers and maintainers, each one needs to be able to install components in the Component Repository. This is a complex procedure which alters the state of the entire cache, therefore it is necessary to lock the cache during each execution of the Component Installation Routine. This may make the Component Repository noticeably inaccessible to other developers and to all users, so there is some risk of degrading the service. This gives further motivation to the idea of local caching.

4 Related Work

The model in the last section was built using Alloy [16], a language for building and analysing software systems. There are several different modelling approaches we could have chosen including [5, 15]. We chose to use Alloy because the models one can write are at the same level of abstraction as our models and there is tool support for analysing them. The models are declarative and describe the structure of systems.

The work described here is a natural extension of work on Java done in collaboration with Drossopoulou and Wragg [26, 25]. Drossopoulou then went on to model dynamic linking in [6] and we looked at the nature of dynamic linking in Java [9]. More recently with Drossopoulou and Lagorio [24] there has been work on providing an operational semantics model for dynamic linking flexible enough to model either Java or $C\#$. There has also been work looking at the software engineering aspects of dynamic linking. We have examined the problems associated with the evolution of Java distributed libraries [10, 11], have looked at the problems that arise with binary compatible code changes in [10] and have built tools, described in [11, 27]. Other formal work on distributed versioning has been done by Sewell in [23].

Rausch [3] models software evolution in terms of Requirements/Assurances contracts. Associated with every component there are explicit textual declarations of the services that the component requires and provides (assures) together with predicates to specify the desired behaviour. The Requirements/Assurances contracts are additional documents that map some or all of the Requirements of one component to the Assures of a second one. As the components evolve, the contract provides the means to check syntactic and behavioral consistency. Since the contracts must explicitly identify the components, it does not seem possible to automatically maintain multiple versions of a single component.

Other groups have studied the problem of protecting clients from troublesome library modifications. [22] identified four problems with ‘parent class exchange’. One of these concerned the introduction of a new (abstract) method into an interface. The other issues all concern library methods which are overridden by client methods in circumstances where, under evolution, the application behaviour is adversely affected. To solve these problems, *reuse contracts* are proposed in order to document the library developer’s design commitments. As the library evolves, the terms of the library’s contract change and the same is true of the corresponding client’s contract. Comparison of these contracts can serve to identify potential problems.

Mezini [19] investigated the same problem (here termed horizontal evolution) and considered that conventional composition mechanisms were not sophisticated enough to propagate design properties to the client. She proposed a *smart* composition model wherein, amongst other things, information about the library calling structure is made available at the client’s site. Successive versions of this information can be compared using reflection to determine how the client can be protected. These ideas have been implemented as an extension to Smalltalk.

Formal treatments of static linking were suggested in [4]. Dynamic linking at a fundamental level has been studied in [13, 1, 28], allowing for modules as first class values, usually untyped, concentrating on confluence and optimization issues. [14], discuss dynamic linking of native code as an extension of Typed Assembly Language without expanding the trusted computing base, while [7] takes a higher-level view and suggests extensions of Typed Assembly Language to support type safe dynamic linking of modules and sharing.

5 Conclusions

Perhaps DLL Hell is a special place reserved only for Microsoft people, but the Upgrade Problem has to be faced by everybody who writes or uses component-based software. Some part of the solution may lie in clever language design, and another may lie in employing strict software development and maintenance procedures – but a large part lies in the versioning strategy of the linking mechanism deployed by the run-time system.

In this paper we have looked at this mechanism as deployed by current .NET development tools. We have shown that if a discipline is kept over which components can be added to the Global Assembly Cache, DLL Hell can be avoided. All the features needed to avoid the problem exist in the .NET manifests but there currently does not seem to be the will to implement the discipline. For example, the multi-part version numbers give developers scope to classify the effects of modifications precisely, and the assembly metadata with the managed code API offers the prospect of reflective code compatibility analysis. However, some of the implementation trade-offs in the current release have frustratingly limited the scope of what is possible. Consequently, the GAC is not usable as a cache for evolving components. As a consequence we have developed our prototype tool, to work alongside and within .NET and to demonstrate what could be attainable.

Our tool is currently more restrictive then is necessary. In the future we will seek to integrate our tool more closely with the GAC and to relax the requirements that components can only be added singly.

Acknowledgements

We would like to thank Sophia Drossopoulou for her careful reading of earlier versions of the model and Michael Huth for suggesting Alloy. Matthew Smith and Rob Chatley made helpful suggestions about the paper. Dejavue.NET was influenced by DEJaVU a tool for Java evolution, implemented by Shakil Shaikh and Miles Barr. Andrew Tseng and Dave Porter of UBS Warburg provided motivation beyond idle academic curiosity. Finally, we would like to thank Sophia Drossopoulou and all the project students of our weekly meeting group for their helpful suggestions and insights while the ideas were being developed.

References

1. Davide Ancona and Elena Zucca. A Primitive calculus for module systems. In *PPDP Proceedings*, September 1999.
2. R. Anderson. The End of DLL Hell. http://msdn.microsoft.com/, January 2000.
3. A. Rausch. *Software Evolution in Componentware using Requirements/Assurances Contracts*, pages 147–156. ACM Press, Limerick, Ireland, May 2000.
4. Luca Cardelli. Program Fragments, Linking, and Modularization. In *POPL'97 Proceedings*, January 1997.

5. A. Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, 1994.
6. S. Drossopoulou. An Abstract Model of Java Dynamic Linking, Loading and Verification. In *Types in Compilation*, September 2001.
7. Dominic Duggan. Sharing in Typed Module Assembly Language. In *Preliminary Proceedings of the Third Workshop on Types in Compilation (TIC 2000)*. Carnegie Mellon, CMU-CS-00-161, 2000.
8. D. Jackson, I. Schechter, and I. Shlyakhter. *Alcoa: the Alloy Constraint Analyzer*, pages 730–733. ACM Press, Limerick, Ireland, May 2000.
9. S. Eisenbach and S. Drossopoulou. Manifestations of the Dynamic Linking Process in Java. http://www-dse.doc.ic.ac.uk/projects/ slurp/dynamic-link/linking.htm, June 2001.
10. S. Eisenbach and C. Sadler. Ephemeral Java Source Code. In *IEEE Workshop on Future Trends in Distributed Systems*, December 1999.
11. S. Eisenbach and C. Sadler. Changing Java Programs. In *IEEE Conference in Software Maintenance*, November 2001.
12. S. Eisenbach. Alloy Model of .NET Evolution. Technical report, http://www.doc.ic.ac.uk/˜sue/alloymodel, August 2003.
13. Kathleen Fisher, John Reppy, and Jon Riecke. A Calculus for Compiling and Linking Classes. In *ESOP Proceedings*, March 2000.
14. Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and Flexible Dynamic Linking of Native Code. In *Preliminary Proceedings of the Third Workshop on Types in Compilation (TIC 2000)*. Carnegie Mellon, CMU-CS-00-161, 2000.
15. G. Holzmann. The Model Checker Spin. In *IEEE Transactions on Software Engineering*, volume 23, 5, May 1997.
16. D. Jackson. Micromodels of Software: Lightweight Modelling and Analysis with Alloy. Technical report, http://sdg.lcs.mit.edu/˜dng/, February 2002.
17. V. Jurisic. Deja-vu.NET: A Framework for Evolution of Component Based Systems. http://www.doc.ic.ac.uk/˜ajf/Teaching/Projects/DistProjects.html, June 2002.
18. E. Meijer and C. Szyperski. Overcoming independent extensibility challenges. *Communications of the ACM*, 45(10):41–44, October 2002.
19. M. Mezini and K. J. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *Proc. of OOPSLA*, pages 97–116, 1998.
20. M. Pietrek. Avoiding DLL Hell: Introducing Application Metadata in the Microsoft .NET Framework. In *MSDN Magazine*, http://msdn.microsoft.com/, October 2000.
21. S. Pratschner. Simplifying Deployment and Solving DLL Hell with the .NET Framework. In *MSDN Magazine*, http://msdn.microsoft.com/, November 2001.
22. K. Mens P. Steyaert, C. Lucas and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. In *Proc. of OOPSLA*, 1996.
23. P. Sewell. Modules, Abstract Types, and Distributed Versioning. In *Proc. of Principles of Programming Languages*. ACM Press, January 2001.
24. S. Drossopouloum, G. Lagorio and S.Eisenbach. Flexible Models for Dynamic Linking. In *Proc. of the European Symposium on Programming*. Springer-Verlag, March 2003.
25. D. Wragg S. Drossopoulou and S. Eisenbach. What is Java binary compatibility? In *Proc. of OOPSLA*, volume 33, pages 341–358, 1998.
26. S. Eisenbach S. Drossopoulou and D. Wragg. A Fragment Calculus: Towards a Model of Separate Compilation, Linking and Binary Compatibility. In *Logic in Computer Science*, pages 147–156, 1999.
27. S. Eisenbach, C. Sadler and S. Shaikh. Evolution of Distributed Java Programs. In *IFIP/ACM Working Conf on Component Deployment*, June 2002.
28. Joe Wells and Rene Vestergaard. Confluent Equational Reasoning for Linking with First-Class Primitive Modules. In *ESOP Proceedings*, March 2000.

On Mobility Extensions of UML Statecharts. A Pragmatic Approach[⋆]

Diego Latella and Mieke Massink

CNR/ISTI – A. Faedo, Via Moruzzi 1, I56124 Pisa, Italy
{d.latella,m.massink}@cnuce.cnr.it

Abstract. In this paper an extension of a behavioural subset of UML Statecharts for modeling mobility issues is proposed. In this extension we relax the unique association between each Statechart - in a collection of Statecharts modeling a system - and its input-queue and we allow the use of (queue) name variables in communication actions. The resulting communication paradigm is much more flexible than the standard asymmetric one and is well suited for the modelling of mobility-oriented as well as fault tolerant systems.

1 Introduction

The Unified Modelling Language (UML) is a graphical modelling language for object-oriented software and systems [12][1]. It has been specifically designed for visualizing, specifying, constructing and documenting several aspects of - or views on - systems. In this paper we concentrate on a behavioural subset of UML Statecharts (UMLSCs) and in particular on a simple but powerful extension of this notation in order to deal with a notion of mobility which can be modeled by the use of a dynamic communication structure and which is sometimes referred to as *mobile computing* (as opposed to *mobile computation*) [1]. In [4] μCharts have been introduced together with their formal semantics[2]. Briefly, a μChart models the behaviour of a system and is a *collection* of UMLSCs, each UMLSC being uniquely associated with its input queue. The computational model of μCharts is an interleaving one with an asynchronous/asymmetric/static pattern of communication. The semantics of a μChart is a Labelled Transition System (LTS) where each state corresponds to the tuple of statuses of the component UMLSCs, each status being composed by the current configuration and the current input queue of the component UMLSC. A transition in the LTS models a step-transition of a component UMLSC. Each step transition corresponds to the selection of an event from the input queue of the component and to the parallel firing of a maximal set of non-conflicting transitions of such component,

⋆ This work has been carried out in the context of Project EU-IST IST-2001-32747 Architectures for Mobility (AGILE),
http://www.pst.informatik.uni-muenchen.de/projekte/agile/.

[1] Although we base our work on UML 1.3, the main features of the notation of interest for our work did not change in later versions.

[2] Note that the name 'μCharts' is used also in [13], but with a different meaning.

which are enabled in its current configuration by the selected event and which do not violate transition priority constraints. The firing of a transition implies also the execution of the output actions associated with such a transition. An output action consists of an *output event* to be sent and the specification of a *destination* component (queue) to which it must be delivered. Its execution consists in delivering the event to the destination queue. Thus the pattern of communication is *asynchronous*, via the input queues, and *asymmetric* because while the sender component specifies to which destination component an event must be addressed, a destination component cannot choose from which sender component to receive input events; it simply *has* to receive any event which has been delivered to its input queue, possibly producing no reaction to such trigger event. This is quite a common situation in the realm of object-oriented notations. There are important features of mobile systems that cannot be expressed directly using only an asymmetric style of communication. In fact there are situations in which we want to make a receiver be able to get input events from *more than one* queue, *explicitly* choosing *when* to receive events from *which* queue. An example of the need of such pattern of communication is, a. o., the Hand-over protocol for mobile telephones, which we shall deal with in the present paper. Moreover, it is well-known [2] that patterns of communication which allow the explicit and dynamic choice of the input queue/entity are essential for the development of fault-tolerant systems since they contribute to fulfilling well established entity isolation principles of error confinement much better than asymmetric patterns. Thus, the extension we propose in this paper consists in letting the *trigger event* specification of transition labels be equipped with the explicit reference to the queue from which the event should be taken. Moreover, such a reference can also be a queue name variable, thus allowing more dynamicity in the choice of the queue(s) from which events are to be received by a UMLSC. Queue names can thus be communicated around and assigned to variables in a way which resembles the $\pi-$calculus [11], although in the more imperative-like framework of UMLSCs.

We are not aware of any other work on extensions of UMLSCs formal semantics with notions of mobile computing, like dynamic addressing, and name-passing. For what concerns formal semantics of UMLSCs numerous contributions can be found in the literature. For a discussion on such contributions and a comparison with our overall approach to UMLSCs semantics we refer the interested reader to [4]. More recently, an SOS approach to (single) UMLSCs formal semantics has been proposed in [15], which is partially based on [7, 10] and assumes the standard UML single input queue per statechart paradigm. Some issues related to communication concepts for (single) Harel statecharts are investigated in [14, 13]. The main focus there is the restriction of classical statechart broadcast by means of explicit feedback interfaces rather than the issues of input queue selection and dynamic addressing in UML statecharts, which we are interested in, in the present paper.

The paper is organized as follows: in Sect. 2 μCharts are briefly recalled and some basic definitions are given. Sect. 3 describes the extension we propose,

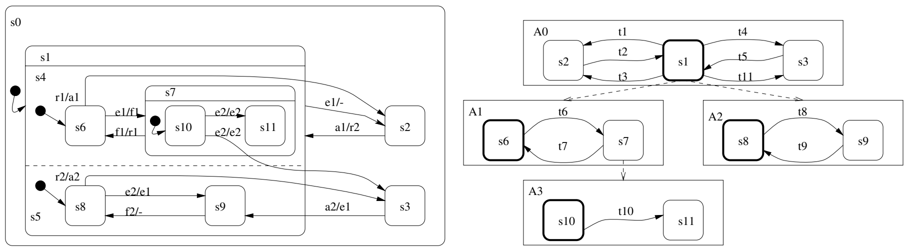


Fig. 1. A UMLSC and its HA.

Table 1. Transition Labels for the HA of Fig. 1.

t	$t1$	$t2$	$t3$	$t4$	$t5$	$t6$	$t7$	$t8$	$t9$	$t10$	$t11$
$SR\ t$	$\{s6\}$	$\emptyset$	$\emptyset$	$\{s8\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{s10\}$
$EV\ t$	$r1$	$a1$	$e1$	$r2$	$a2$	$e1$	$f1$	$e2$	$f2$	$e2$	$e2$
$AC\ t$	$a1$	$r2$	ϵ	$a2$	$e1$	$f1$	$r1$	$e1$	ϵ	$e2$	$e2$
$TD\ t$	$\emptyset$	$\{s6, s8\}$	$\emptyset$	$\emptyset$	$\{s6, s9\}$	$\{s10\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

including its formal semantics definition. The Hand-over protocol specification example is given in Sect. 4. Finally in Sect. 5 some conclusions are drawn and directions for further work are sketched. Detailed proofs relevant to the present paper can be found in [9].

2 μCharts

In this section the basic definitions related to μCharts are briefly recalled. They are treated in depth in [4] where the interested reader is referred to. We use Hierarchical Automata (HAs) [10] as the abstract syntax for UMLSCs. HAs are composed of simple sequential automata related by a *refinement function.* In [7] an algorithm for mapping a UMLSC to a HA is given; the reader interested in its technical details is referred to the above mentioned paper. Here we just recall the main ingredients of this mapping, by means of a simple example. Consider the UMLSC of Fig.1 (left). Its HA is shown on the right side of the figure. Roughly speaking, each OR-state of the UMLSC is mapped into a sequential automaton of the HA while basic and AND-states are mapped into states of the sequential automaton corresponding to the OR-state immediately containing them. Moreover, a refinement function maps each state in the HA corresponding to an AND-state into the set of the sequential automata corresponding to its component OR-states. In our example (Fig.1, right), OR-states $s0, s4, s5$ and $s7$ are mapped to sequential automata $A0, A1, A2$ and $A3$, while state $s1$ of $A0$, corresponding to AND-state $s1$ of our UMLSC, is refined into $\{A1, A2\}$. Non-interlevel transitions are represented in the obvious way: for instance transition $t8$ of the HA represents the transition from state $s8$ to state $s9$ of the UMLSC. The labels of transitions are collected in Table 1; for example the *trigger event* of $t8$, namely $EV\ t8$, is $e2$ while its associated *output event*, namely $AC\ t8$ is $e1$. An interlevel transition is represented as a transition t departing from (the HA state corresponding to) its highest source and pointing to (the HA state corresponding to) its highest target. The set of the other sources, resp., targets, are recorded in the *source restriction* - $SR\ t$, resp. *target determinator* $TD\ t$, of t. So, for instance, $SR\ t1 = \{s6\}$ means that a necessary condition for $t1$ to be enabled is that the current state configuration contains not only $s1$ (the source of $t1$), but *also* $s6$. Similarly, when firing $t2$ the new state configuration will contain $s6$ and $s8$, besides $s1$. Finally, each transition has a guard $G\ t$, not shown in this example. The structure of transition labels will be properly accommodated later on in this paper in order to support the mobility extensions. Transitions originating from the same state are said to be in *conflict*. The notion of *conflict* between transitions needs to be extended in order to deal with state hierarchy. When transitions t and t' are in conflict we write $t\#t'$. The complete formal definition of conflict for HAs can be found in [7, 4] where also the notion of *priority* for (conflicting) transitions is defined. Intuitively transitions coming from deeper states have higher priority. For the purposes of the present paper it is sufficient to say that priorities form a partial order. We let πt denote the priority of transition t and $\pi t \sqsubseteq \pi t'$ mean that t has lower priority than (the same priority as) t'. In the sequel we will be concerned only with HAs.

A μChart is a collection of UML Statecharts (actually HAs) communicating via input queues. We consider a restricted subset of UML Statcharts, which, nevertheless includes all the interesting conceptual issues related to concurrency in the dynamic behaviour, like sequentialisation, non-determinism and parallelism. We call such a subset a "Behavioural subset of UML Statecharts", UMLSCs in short. More specifically, we do not consider history, action and activity states; we restrict events to signal ones without parameters (actually we do not interpret events at all); time and change events, object creation and destruction events, and deferred events are not considered as are branch transitions; for the sake of simplicity, given that in the present paper our main focus is on the manipulation of queues for achieving dynamic communication structures, we restrict data values to a single type, namely *queue names*. We also abstract from entry and exit actions of states. The interested reader can find a complete discussion on the above choices together with their motivations in [4].

2.1 Basic Definitions

The first notion we need to define is that of (sequential) automaton[3].

[3] In the following we shall freely use a functional-like notation in our definitions where: (i) currying will be used in function application, i.e. $f\ a_1\ a_2 \dots\ a_n$ will be used instead of $f(a_1, a_2, \dots, a_n)$ and function application will be considered left-associative; (ii) for function $f : X \to Y$ and $Z \subseteq X$, $f\ Z = \{y \in Y \mid \exists x \in Z.\ y = fx\}$, $rng\ f$ denotes the *range* of f and $f_{|Z}$ is the restriction of f to Z. (iii) by $\exists_1 x.\ P\ x$ we mean "there exists a unique x such that $P\ x$". Finally, for set X, we let X^* denote the set of finite sequences over D. The empty sequence will be denoted by ϵ and the concatenation of sequnce x with sequence y will be indicated by xy. For sequences x, y and z we let predicate $\mathsf{mrg}\ x\ y\ z$ hold iff z is a non-deterministic merge of x and y, that is z is a permutation of xy such that the occurrence order in x (resp. y) of the elements of x (resp. y) is preserved in z; its extension $\mathsf{mrg}_{j=1}^{n}\ x_j\ z$ to n sequences is defined in the obvious way.

Definition 1 (Sequential Automata). *A sequential automaton A is a 4-tuple $(\sigma_A, s^0_A, \lambda_A, \delta_A)$ where σ_A is a finite set of* states *with $s^0_A \in \sigma_A$ the* initial state*, λ_A is a finite set of* transition labels *and $\delta_A \subseteq \sigma_A \times \lambda_A \times \sigma_A$ is the* transition relation.

The labels in λ_A have a particular structure as we briefly mentioned above and we shall discuss later in more detail. Moreover, we assume that all transitions are uniquely identifiable. This can be easily achieved by just assigning them arbitrary unique names, as we shall do throughout this paper. For sequential automaton A let functions $SRC, TGT : \delta_A \to \sigma_A$ be defined as $SRC(s, l, s') = s$ and $TGT(s, l, s') = s'$. Let $\mathcal{N}$ be a set of (queue) *names*. HAs are defined as follows:

Definition 2 (Hierarchical Automata). *A HA H is a 3-tuple (F, E, ρ) where F is a finite set of sequential automata with mutually disjoint sets of states, i.e. $\forall A_1, A_2 \in F.\ \sigma_{A_1} \cap \sigma_{A_2} = \emptyset$ and E is a finite set of* events*, with $E \subseteq \mathcal{N}$; the* refinement function *$\rho : \bigcup_{A \in F} \sigma_A \to 2^F$ imposes a tree structure to F, i.e. (i) there exists a unique root automaton $A_{root} \in F$ such that $A_{root} \notin \bigcup rng\ \rho$, (ii) every non-root automaton has exactly one ancestor state: $\bigcup rng\ \rho = F \setminus \{A_{root}\}$ and $\forall A \in F \setminus \{A_{root}\}.\ \exists_1 s \in \bigcup_{A' \in F \setminus \{A\}} \sigma_{A'}.\ A \in (\rho\ s)$ and (iii) there are no cycles: $\forall S \subseteq \bigcup_{A \in F} \sigma_A.\ \exists s \in S.\ S \cap \bigcup_{A \in \rho s} \sigma_A = \emptyset$.*

We say that a state s for which $\rho\ s = \emptyset$ holds is a *basic* state. From the above definition the reader can see that the only type of events we deal with are queue names, as mentioned above. Every sequential automaton $A \in F$ characterizes a HA in its turn: intuitively, such a HA is composed by all those sequential automata which lay below A, including A itself, and has a refinement function ρ_A which is a restriction of ρ:

Definition 3. *For $A \in F$ the* automata*,* states *and* transitions *under A are defined respectively as $\mathcal{A}\ A \ \triangleq\ \{A\} \cup (\bigcup_{A' \in \left(\bigcup_{s \in \sigma_A} (\rho_A s)\right)} (\mathcal{A}\ A'))$, $\mathcal{S}\ A \ \triangleq\ \bigcup_{A' \in \mathcal{A}\ A} \sigma_{A'}$, and $\mathcal{T}\ A \ \triangleq\ \bigcup_{A' \in \mathcal{A}\ A} \delta_{A'}$*

The definition of sub-hierarchical automaton follows:

Definition 4 (Sub-Hierarchical Automata). *For $A \in F$, (F_A, E, ρ_A), where $F_A \ \triangleq\ (\mathcal{A}\ A)$, and $\rho_A \ \triangleq\ \rho_{|(\mathcal{S}\ A)}$, is the HA characterized by A.*

In the sequel for $A \in F$ we shall refer to A both as a sequential automaton and as the sub-hierarchical automaton of H it characterizes, the role being clear from the context. H will be identified with A_{root}. Sequential Automata will be considered a degenerate case of HAs. μCharts are defined as follows:

Definition 5 (μChart). *A μChart is a tuple $(E, H_1, \ldots, H_k)$ where (i) $E \subseteq \mathcal{N}$, (ii) $H_j = (F_j, E, \rho_j)$ is a HA for $j = 1 \ldots k$ and (iii) $\mathcal{S}\ H_j \cap \mathcal{S}\ H_i = \emptyset$ for $i \neq j$.*

The complete formal semantics for μCharts can be found in [4]. A μChart is mapped into a LTS. Each state of such LTS is a tuple of configuration-queue pairs; each pair records the current configuration and the current input queue of a distinct HA in the μChart. The transition relation of the LTS is defined by means of a derivation system composed by four rules. One of them, the *top rule*, defines the transition relation and uses in turn an auxiliary relation defined by the three remaining rules, the so called *core semantics*. In the next sections we shall deal with the semantics of the *extension* of μCharts we propose. Before proceeding with the semantics definitions we need a few more concepts:

Definition 6 (Configurations). *A* configuration *of HA $H = (F, E, \rho)$ is a set $\mathcal{C} \subseteq (\mathcal{S}\ H)$ such that (i) $\exists_1 s \in \sigma_{A_{root}}.\ s \in \mathcal{C}$ and (ii) $\forall s, A.\ s \in \mathcal{C} \wedge A \in \rho\ s \Rightarrow \exists_1 s' \in \sigma_A.\ s' \in \mathcal{C}$*

A *configuration* denotes a global state of a HA, composed of local states of component sequential automata. For $A \in F$ the set of all configurations of A is denoted by Conf_A. In the extension we propose we will deal with queue name variables. Thus we assume a universe $\mathcal{V}$ of such variables, with $\mathcal{N} \cap \mathcal{V} = \emptyset$. We also need communication packets; a packet is a pair *(destination, message-body)*. The set $\mathcal{P}$ of *packets* is defined as $\mathcal{P} \stackrel{\Delta}{=} \mathcal{N} \times \mathcal{N}$, that is message bodies can be only queue names. Due to the presence of variables we need also packet *terms* and stores. The set $\mathcal{P}t$ of packet terms is defined as $\mathcal{P}t \stackrel{\Delta}{=} (\mathcal{N} \cup \mathcal{V}) \times (\mathcal{N} \cup \mathcal{V})$. Stores are defined as follows:

Definition 7 (Stores). *A* store *β is a partial function $\beta : \mathcal{V} \to \mathcal{N}$. As usual $\beta\ v = \perp$ means that v is not bound by β to any value, namely β is undefined on v. We let $\perp\!\!\perp$ be the function such that $\perp\!\!\perp\ v = \perp$ for all $v \in \mathcal{V}$. For store β we let $\hat{\beta}$ denote its extension to names and packet terms, in the usual way: $\hat{\beta}\ q \stackrel{\Delta}{=} q$, if $q \in \mathcal{N}$, $\hat{\beta}\ v \stackrel{\Delta}{=} \beta\ v$, if $v \in \mathcal{V}$, and $\hat{\beta}\ (d, b) \stackrel{\Delta}{=} (\hat{\beta}\ d, \hat{\beta}\ b)$. For $v \in \mathcal{V}$ and $q \in \mathcal{N}$ the unit store $[v \mapsto q]$ is the function such that $[v \mapsto q]v' \stackrel{\Delta}{=} q$, if $v' = v$, and $[v \mapsto q]v' \stackrel{\Delta}{=} \perp$, if $v' \neq v$. Finally, for stores β_1 and β_2 we let store $\beta_1 \triangleleft \beta_2$ be the function such that $(\beta_1 \triangleleft \beta_2)\ v \stackrel{\Delta}{=} \beta_2\ v$, if $\beta_2\ v \neq \perp$ and $(\beta_1 \triangleleft \beta_2)\ v \stackrel{\Delta}{=} \beta_1\ v$, if $\beta_2\ v = \perp$.*

In the following, we shall often consider stores as sets of pairs and compose them using set-union, when the domains of the component functions are mutually disjoint. While in classical statecharts the environment is modelled by a set, in the official definition of UMLSCs the particular nature of the environment is not specified. Actually it is stated to be a *queue*, the *input queue*, but the management policy of such a queue is not defined. We choose *not* to fix any particular semantics such as a set, or a multi-set or a FIFO queue etc., but to model the input queue in a policy-independent way, freely using a notion of abstract data types. In the following we assume that for set D, Θ_D denotes the set of all structures of a certain kind (like FIFO queues, or multi-sets, or sets) over D and we assume to have basic operations for inserting and removing elements from such structures. Among such operations, the predicate $(Sel\ \mathcal{D}\ d\ \mathcal{D}')$ which states that $\mathcal{D}'$ is the structure resulting from selecting d from $\mathcal{D}$, is of particular importance in the context of the present paper. Of course, the selection policy depends on the choice for the particular semantics. In the present paper we assume that if $\mathcal{D}$ is the empty structure, *nil* then $(Sel\ \mathcal{D}\ d\ \mathcal{D}')$ is false for all d and $\mathcal{D}'$. In the sequel we shall often speak of the *input queue* or simply *queue* meaning by that a structure in Θ_D, for proper D, and abstracting from its particular semantics.

3 μCharts with Explicit Dynamic Channels

In this section we describe our mobility extension of μCharts. As before, a system is modeled by a fixed collection of UMLSCs (actually HAs), but now each HA can be associated to several *input-queues.* The association is specified by explicit reference, in any transition of the HA, to the input queue from which the trigger event of that transition is to be selected. The association is *dynamic* since, besides queue names, uniquely associated to distinct queues, queue name *variables* can be used as well. Consequently we have to re-define the labels of the transitions of HAs. Let $H = (F, E, \rho)$ be a HA of a μChart S. The label l of transition $t = (s, l, s') \in \delta_A$, for $A \in F$, is the tuple $(SR\ t, IQ\ t, EV\ t, G\ t, DQ\ t, AC\ t, TD\ t)$. The meaning of $SR\ t, G\ t$ and $TD\ t$ is the same as briefly discussed in Sect. 2. The specification of the queue from which the *trigger event* $EV\ t$ should be selected is given by the *input-queue* component of the label, $IQ\ t$. In the present paper, for the sake of simplicity, the only kind of actions a HA can perform when firing a transition is the sending of an event to a queue. Consequently, on the output side, the specification of the *destination queue* $DQ\ t$ is added to that of the *output event* $AC\ t$. At the concrete syntax level, the label of a transition t of a UMLSC will have the form $q?e/q'!e'$ where $IQ\ t = q$, $EV\ t = e$, $DQ\ t = q'$ and $AC\ t = e'$ [4]. As we said above the only values we are dealing with are queue-names[5] and we allow the use of variables in order to express dynamic addresses. Consequently the trigger event, the input-queue, the output event and the destination queue can be queue-names or queue-name variables, i.e. $EV\ t, IQ\ t, DQ\ t, AC\ t \in E \cup \mathcal{V}$. Furthermore, the above assumptions imply that there is a pool of "shared" queues through which the HAs communicate. We call such a pool a *multi-queue* and a collection of HAs communicating via a multi-queue is called a $\delta\mu$Chart. Multi-queues are an extension of input-queues. We need to redefine the selection relation and multi-queue extension. We do this informally as follows: $Sel\ \mathcal{E}\ q\ e\ \mathcal{E}'$ holds iff e is the element selected from queue (named) q of multi-queue $\mathcal{E}$ and $\mathcal{E}'$ is the multi-queue resulting from deleting e from queue (named) q of $\mathcal{E}$. We let $\mathcal{E}[(q, e)]$ denote the multi-queue equal to $\mathcal{E}$ except that on queue (named) q of $\mathcal{E}$ element e is inserted, where $(q, e) \in \mathcal{P}$; for $\mathcal{U} \in \mathcal{P}^*$, $\mathcal{E}[\mathcal{U}]$ is defined in the obvious way: $\mathcal{E}[(q_1, e_1), (q_2, e_2), \ldots, (q_n, e_n)] \stackrel{\Delta}{=} (\mathcal{E}[(q_1, e_1)])[(q_2, e_2), \ldots, (q_n, e_n)]$ where $\mathcal{E}[\epsilon] \stackrel{\Delta}{=} \mathcal{E}$.

[4] Notice that in this paper we use a syntax for event sending (namely exclamation mark) which is slightly different from the standard syntax for method calling (namely a dot). This is only for symmetry with our notation for input (namely question mark) and for our deliberate focusing on semantical more than syntactical issues.

[5] This is a similar situation as in the $\pi - calculus$ [11], although in a completely different context.

The Operational Semantics of $\delta\mu$Chart $S = (E, H_1, \ldots H_k)$ is a transition system. Each global state is a tuple $(\mathcal{E}, Loc_1, \ldots, Loc_k)$. $\mathcal{E}$ is the current value of multi-queue and Loc_j is the current status of HA H_j. The status Loc_j of a component HA H_j is a pair $(\mathcal{C}_j, \beta_j)$ where $\mathcal{C}_j$ is the current configuration of H_j and β_j is the current store of H_j. Each transition corresponds to a step of one component HA H_j. We recall here that in μCharts, being each HA uniquely associated to a distinct queue, the hypothetical scheduler associated to the semantics of state machines in [12] is only left with the job of (i) choosing a HA to execute a step and (ii) selecting an event *in the input queue of the selected HA* to feed into its state machine in order to perform such a step. Notice that the scheduler is somehow "blind" in this job w.r.t. the event: it chooses and *de-queues* an event regardless of whether such an event will be actually used by the state machine, i.e. there are transitions which are enabled by the event. If this is not the case, the state machine stutters and the event is lost (or, at most, deferred[6]). In the case of $\delta\mu$Charts there is no longer a single input queue associated to each HA, but the multi-queue. Thus, in $\delta\mu$Charts the scheduler must also select the *particular input queue* from which the event is to be selected. We shall deal with input queue selection in Sect.3.2, where we define the top rule(s) of the derivation system for the transition relation. Since such a derivation system exploits some properties of the core semantics on which it is based, we prefer to first define, in Sect. 3.1, the core semantics which is obviously parametric w.r.t. the selected queue.

3.1 Core Semantics

The core semantics is given in Fig.2 and has the same structure as that for μCharts. It defines the relation $A \uparrow P :: (\mathcal{C}, \beta) \overset{(q,e)/\mathcal{U}}{\longrightarrow}_L (\mathcal{C}', \beta')$ which models the step-transitions of HA A, and L is the set containing the transitions of A which are fired. In such a relation P is a set of transitions. It represents a constraint on each of the transitions fired in the step, namely that it must not be the case that there is a transition in P with a higher priority. So, informally, $A \uparrow P :: (\mathcal{C}, \beta) \overset{(q,e)/\mathcal{U}}{\longrightarrow}_L (\mathcal{C}', \beta')$ should be read as "A, on configuration and store $(\mathcal{C}, \beta)$, with input event e from queue q can fire the transitions in the set L moving to configuration and store $(\mathcal{C}', \beta')$, producing output $\mathcal{U}$, when required to fire transitions with priorities not smaller than that of any transition in P". Set P will be used to record the transitions a certain automaton can do when considering its sub-automata. More specifically, for sequential automaton A, P will accumulate all transitions which are enabled in the ancestors of A. The Core Semantics definition uses functions LE_A and E_A. For HA A, $\mathsf{LE}_A\ \mathcal{C}\ \beta\ q\ e$ is the set $\{t \in \delta_A \mid \{(SRC\ t)\} \cup (SR\ t) \subseteq \mathcal{C} \wedge \hat{\beta}(IQ\ t) = q \wedge \mathtt{MATCH}\ e\ (EV\ t) \wedge (\mathcal{C}, \beta, e) \models (G\ t)\}$ i.e. the set of those transitions in δ_A, i.e. *local* to the root of A, which are enabled in the current configuration $\mathcal{C}$, store β with input event e from input queue q. Function $\mathtt{MATCH}$ is defined as follows: $\mathtt{MATCH}\ e\ x \overset{\Delta}{=} (x \in \mathcal{V} \vee x = e)$.

[6] We do not deal with deferred events in our current work.

Progress Rule

$$\frac{\begin{array}{l} t \in \mathsf{LE}_A\ \mathcal{C}\ \beta\ q\ e \\ \beta' = \text{if } (EV\ t) \in \mathcal{V} \text{ then } [(EV\ t) \mapsto e] \text{ else } \perp\!\!\!\perp \\ \nexists t' \in P \cup \mathsf{E}_A\ \mathcal{C}\ \beta\ q\ e.\ \pi t \sqsubset \pi t' \end{array}}{A \uparrow P :: (\mathcal{C}, \beta) \overset{(q,e)/(DQ\ t, AC\ t)}{\longrightarrow}{}_{\{t\}} (DST\ t, \beta')}$$

Stuttering Rule

$$\frac{\begin{array}{l} \{s\} = \mathcal{C} \cap \sigma_A \\ \rho_A\ s = \emptyset \\ \forall t \in \mathsf{LE}_A\ \mathcal{C}\ \beta\ q\ e.\ \exists t' \in P.\ \pi t \sqsubset \pi t' \end{array}}{A \uparrow P :: (\mathcal{C}, \beta) \overset{(q,e)/\epsilon}{\longrightarrow}{}_{\emptyset} (\{s\}, \perp\!\!\!\perp)}$$

Composition Rule

$$\frac{\begin{array}{l} \{s\} = \mathcal{C} \cap \sigma_A \\ \rho_A\ s = \{A_1, \ldots, A_n\} \neq \emptyset \\ \left(\bigwedge_{j=1}^n A_j \uparrow P \cup \mathsf{LE}_A\ \mathcal{C}\ \beta\ q\ e :: (\mathcal{C}, \beta) \overset{(q,e)/\mathcal{U}_j}{\longrightarrow}{}_{L_j} (\mathcal{C}_j, \beta_j)\right) \wedge \mathsf{mrg}_{j=1}^n \mathcal{U}_j\ \mathcal{U} \\ \left(\bigcup_{j=1}^n L_j = \emptyset\right) \Rightarrow (\forall t \in \mathsf{LE}_A\ \mathcal{C}\ \beta\ q\ e.\ \exists t' \in P.\ \pi t \sqsubset \pi t') \end{array}}{A \uparrow P :: (\mathcal{C}, \beta) \overset{(q,e)/\mathcal{U}}{\longrightarrow}{}_{\bigcup_{j=1}^n L_j} (\{s\} \cup \bigcup_{j=1}^n \mathcal{C}_j, \bigcup_{j=1}^n \beta_j)}$$

Fig. 2. Core operational semantics rules for $\delta\mu$Charts.

We skip the details of guard evaluation in this paper for lack of space. Function E_A extends LE_A in order to cover *all* the transitions of A including those of sub-automata of A, i.e. $\mathsf{E}_A\ \mathcal{C}\ \beta\ q\ e \overset{\Delta}{=} \bigcup_{A' \in (\mathcal{A}\ A)} \mathsf{LE}_{A'}\ \mathcal{C}\ \beta\ q\ e$. In the Core Semantics, the Progress Rule establishes that if there is a transition of A enabled and the priority of such a transition is "high enough" then the transition fires and a new status is reached accordingly. The store generated contains *only* the information related to the possible binding of $EV\ t$, when the latter is a variable. The global store of the HA which A belongs to will be extended properly by the top-level rules (see below). Notice that the destination $(DQ\ t)$ and the message body $(AC\ t)$ are dealt with in a symbolic way at the Core Semantics level. They will be evaluated by the Global Progress Rule. The reason why the variable evaluation cannot be performed by the core semantics should be clear: a variable used in the output action of a transition t of a parallel component, say A of a HA might be bound by *another* parallel component of the same HA. So the correct store to be used for $(DQ\ t, AC\ t)$ pairs is *not* available when applying the Progress Rule to A [7]. For transition t, $DST\ t$ is defined as the set $\{s \mid \exists s' \in (TD\ t).\ (TGT\ t) \preceq s \preceq s'\}$. Intuitively $DST\ t$ comprises all states which are below $(TGT\ t)$ in the state-hierarchy down to those in $(TD\ t)$. The state preorder $\preceq$ is formally defined e.g. in [4]. The Composition Rule stipulates how automaton A delegates the execution of transitions to its sub-automata and these transitions are propagated upwards. Notice that for all v, i, j, $\beta_i\ v \neq \perp$ and $\beta_j\ v \neq \perp$ implies $\beta_i\ v = \beta_j\ v = e$. Finally, if there is no transition of A enabled with "high enough" priority and moreover no sub-automata exist to which the execution of transitions can be delegated, then A has to "stutter", as enforced by the Stuttering Rule. The following theorem links our semantics to the general requirements set by the official semantics of UML:

[7] In [9] a "late" binding semantics is also provided where the destination and message body are evaluated using the *current* store.

Theorem 1. *Given HA $H = (F, E, \rho)$ element of a $\delta\mu$Chart, for all $A \in F, e \in E, L, \mathcal{C}, \beta, q, \mathcal{U}$ the following holds: $A \uparrow P :: (\mathcal{C}, \beta) \overset{(q,e)/\mathcal{U}}{\longrightarrow}_L (\mathcal{C}', \beta')$ for some $\mathcal{C}', \beta'$ iff L is a maximal set, under set inclusion, which satisfies all the following properties: (i) L is conflict-free, i.e. $\forall t, t' \in L.\ \neg t \# t'$; (ii) all transitions in L are enabled in the current status, i.e. $L \subseteq \mathsf{E}_A\ \mathcal{C}\ \beta\ q\ e$; (iii) there is no transition outside L which is enabled in the current status and which has higher priority than a transition in L, i.e. $\forall t \in L.\ \ \not\exists t' \in \mathsf{E}_A\ \mathcal{C}\ \beta\ q\ e.\ \pi t \sqsubset \pi t'$; and (iv) all transitions in L respect P, i.e. $\forall t \in L.\ \ \not\exists t' \in P.\ \pi t \sqsubset \pi t'$.*

Proof. The proof can be carried out in a similar way as for the main theorem of [4], by structural induction for the direct implication and by derivation induction for the reverse implication.

3.2 Input Queue Selection

The selection from the multi-queue is tightly connected to the possibility of generating unwanted extra-stuttering. For generating a step of a HA H starting from configuration $\mathcal{C}$ and multi-queue $\mathcal{E}$, we consider only those queues in $\mathcal{E}$ which are *relevant* in $\mathcal{C}$; q is relevant in $\mathcal{C}$ if there is a transition the source of which is contained in $\mathcal{C}$ and the label of which uses q as input queue. No stuttering of a HA H in $\mathcal{C}$ and multi-queue $\mathcal{E}$ should be allowed which is caused by the selection of a (relevant) queue q of $\mathcal{E}$ and an event dequeued from it for which no transition is enabled *while* there is another (relevant) queue q' of $\mathcal{E}$ and/or another event such that a transition could be enabled. Thus we allow stuttering *only* on relevant queues and *only* if there is no transition enabled. Notice that a stuttering step modifies the multi-queue in a way which depends on the selected queue. The set $\mathsf{LR}_A\ \mathcal{C}\ \beta$ of *local relevant* and the set $\mathsf{R}_A\ \mathcal{C}\ \beta$ of the *relevant* queues of $A \in H = (F, E, \rho)$ are defined respectively as $\{q \in E \mid \exists t \in \delta_A.\ (SRC\ t) \in \mathcal{C} \wedge \hat{\beta}(IQ\ t) = q\}$ and $\bigcup_{A' \in (\mathcal{A}\ A)} \mathsf{LR}_{A'}\ \mathcal{C}\ \beta$. For $A \in F$ of $HA\ H = (F, E, \rho)$, $\mathcal{C} \in \mathrm{Conf}_A$, store β, multiqueue $\mathcal{E}$ on E, the set $\mathsf{PE}_A\ \mathcal{C}\ \beta\ \mathcal{E}$ of *Potentially Enabled Transitions* of A, on configuration $\mathcal{C}$, store β and multi-queue $\mathcal{E}$ is the set $\{t \in \mathcal{T}\ A \mid \exists q \in \mathsf{R}_A\ \mathcal{C}\ \beta, e \in E.\ Sel\ \mathcal{E}\ q\ e\ \mathcal{E}' \wedge t \in \mathsf{E}_A\ \mathcal{C}\ \beta\ q\ e\}$. Obviously, the fact that a transition $t \in \mathsf{PE}_A\ \mathcal{C}\ \beta\ \mathcal{E}$ will actually be enabled depends on the choice of the particular relevant queue q and the selection of the particular event e from q. The following lemmas relate stuttering with the set of potentially enabled transitions and with the resulting configurations and stores. They will be useful for a better understanding of the top-level rules of the definition of the transition relation. Lemma 1 states that if there is no potentially enabled transition - in a given configuration, store and multi-queue - then every step from that configuration and store involving any relevant queue and any selected event is a stuttering step. Vice-versa, if from a given configuration, store and multi-queue only stuttering steps are possible, whatever choice is done for the queue and the input event, then there are no potentially enabled transitions. The lemma easily follows from from Lemma 3 in [9]. Lemma 2 guarantees that stuttering does not change the store and the configuration. Its proof is similar that of Lemma A.1 (ii) in [8].

Lemma 1.
For all $H = (F, E, \rho), \mathcal{C} \in \mathrm{Conf}_H$, *store* $\beta, \mathcal{E}$ *multiqueue on* E *the following holds: (i) for all* $q, e \in E, \mathcal{E}'$ *multiqueue on* E*: if* $\mathsf{PE}_H\ \mathcal{C}\ \beta\ \mathcal{E} = \emptyset$ *and* $q \in \mathsf{R}_H\ \mathcal{C}\ \beta$*, and* $Sel\ \mathcal{E}\ q\ e\ \mathcal{E}'$ *and* $H \uparrow \emptyset :: (\mathcal{C}, \beta) \xrightarrow{(q,e)/\mathcal{U}}_L (\mathcal{C}', \beta')$ *for some* $\mathcal{C}', \beta', L$*, then* $L = \emptyset$*; (ii) if for all* $q, e \in E, \mathcal{E}'$ *multiqueue on* E *such that* $Sel\ \mathcal{E}\ q\ e\ \mathcal{E}'$, $\mathcal{C}', \beta'$ *we have* $H \uparrow \emptyset :: (\mathcal{C}, \beta) \xrightarrow{(q,e)/\mathcal{U}}_\emptyset (\mathcal{C}', \beta')$ *then we also have* $\mathsf{PE}_A\ \mathcal{C}\ \beta\ \mathcal{E} = \emptyset$

Lemma 2.
For all $H = (F, E, \rho), \mathcal{C} \in \mathrm{Conf}_H$, *store* $\beta, q, e \in E$ *if* $H \uparrow \emptyset :: (\mathcal{C}, \beta) \xrightarrow{(q,e)/\mathcal{U}}_\emptyset (\mathcal{C}', \beta')$ *then* $\mathcal{C}' = \mathcal{C}$ *and* $\beta' = \perp\!\!\!\perp$.

Global Progress rule

$$\frac{\begin{array}{lr}
q \in \mathsf{R}_{H_j}\ \mathcal{C}_j\ \beta_j & (1)\\
Sel\ \mathcal{E}\ q\ e\ \mathcal{E}' & (2)\\
H_j \uparrow \emptyset :: (\mathcal{C}_j, \beta_j) \xrightarrow{(q,e)/\mathcal{U}_j}_{L_j} (\mathcal{C}_j', \beta_j') & (3)\\
L_j \neq \emptyset & (4)\\
\mathcal{U}_j' = map\ (\widehat{\beta_j \lhd \beta_j'})\ \mathcal{U}_j & (5)
\end{array}}{\begin{array}{l}
(\mathcal{E}, (\mathcal{C}_1, \beta_1), .., (\mathcal{C}_j, \beta_j), .., (\mathcal{C}_k, \beta_k)) \longrightarrow\\
(\mathcal{E}'[\mathcal{U}_j'], (\mathcal{C}_1, \beta_1), .., (\mathcal{C}_j', \beta_j \lhd \beta_j'), .., (\mathcal{C}_k, \beta_k))
\end{array}}$$

Global Stuttering rule

$$\frac{\begin{array}{lr}
q \in \mathsf{R}_{H_j}\ \mathcal{C}_j\ \beta_j & (1)\\
Sel\ \mathcal{E}\ q\ e\ \mathcal{E}' & (2)\\
\mathsf{PE}_H\ \mathcal{C}_j\ \beta_j\ \mathcal{E} = \emptyset & (3)
\end{array}}{\begin{array}{l}
(\mathcal{E}, (\mathcal{C}_1, \beta_1), .., (\mathcal{C}_j, \beta_j), .., (\mathcal{C}_k, \beta_k)) \longrightarrow\\
(\mathcal{E}', (\mathcal{C}_1, \beta_1), .., (\mathcal{C}_j, \beta_j), .., (\mathcal{C}_k, \beta_k))
\end{array}}$$

Fig. 3. $\delta\mu$Charts Transition Relation definition.

The definition of the transition relation for $\delta\mu$Charts is given in Fig. 3. It is composed of two top-level rules. The Global Progress rule produces all non-stuttering steps (premise (4)). Notice that only relevant queues are considered (premise (1)); in fact non relevant queues would generate (undesired) stuttering, as can be derived from the definitions. As usual, higher order function *map* applied on function β and sequence $\mathcal{U}$ returns the sequence obtained by applying β to each and every element of $\mathcal{U}$ (premise (5)). The Global Stuttering rule takes care of stuttering. When there are no potentially enabled transitions (premise (3)), from Lemma 1 (i) we know that *only* stuttering can occur; moreover, from (ii) of the same lemma we know that *any* stuttering situation will require that no transition is potentially enabled. Notice that also in this rule we restrict to *relevant* queues (premise (1)): we restrict to those stuttering steps which involve relevant queues. Should we have chosen a queue which is not relevant, we would have ended up with a step leading to a multi-queue where an element of such a non relevant queue (premise (2)) would have been removed. We consider this undesired behaviour (a too much blind scheduler!). Finally, the choice of different relevant queues produces different stuttering steps, due to different multi-queues in the next global state (premise (2) and consequent); on the other hand, all configurations and store remain unchanged, including those of the HA which generated the stuttering step (Lemma 2).

4 Example: The Hand-Over Protocol

In this section we model the Hand-over protocol for mobile phones as described in [3]. The example scenario consists of a mobile station, a switching center and two base stations. The mobile station is mounted in a car moving through two different geographical areas (cells) and provides services to an end user; the switching center is the controller of the radio communications within the whole area composed by the two cells; the two base stations, one for each cell, are the interfaces between the switching center and the mobile station. The switching center receives messages addressed to the car (user) from the external environment and forwards them to the base station of the cell where the car currently resides. The base station of such cell forwards these messages to the car which presents them to the user. The communication between the base station and the car takes place via a private data channel shared with the car while the latter is in the related cell. As soon as the switching center is signalled of the fact that the car is leaving the current cell and entering the other one, it sends the base station of the current cell the control information necessary to the car in order to get connected to the base station of the other cell. The base station of the current cell forwards this information to the car using a private control channel shared with the car while the latter is in the related cell. The above mentioned information consists of the references to the data channel and the control channel of the other base station. Finally the current base station acknowledges the switching center and waits idle to be resumed by the latter. Once the car received the control information mentioned above, it uses it for getting messages from the other cell. Once the switching center received the acknowledgement from the base station of the current cell it wakes up the other one. At this point the flow of messages from the external environment to the car (user) continues, but using the base station of the other cell. The complete $\delta\mu$Chart of the protocol is given in Fig. 4. We use the convention of writing queue names in upper-case characters, while variables are written in lower-case characters[8]. We abstract from the user, which we actually consider part of the environment; so, the messages received by the car are actually sent out/back to the environment. Moreover, we abstract from the real content of the messages; we use a single value MSG for representing any message. We also abstract from the details by which the switching center is notified that the car moved from one cell to the other; we model this situation in the environment using non-determinism. Finally we assume that initially the car resides in the cell associated to the first base station. The elements of the multi-queue are assumed to be FIFO queues. The alphabet of the $\delta\mu$Chart is the set of all queue names appearing in the Statecharts in the figure. The only name which does not correspond to a queue in the multi-queue is obviously MSG. Initially, all queues are empty

[8] Notice that in the example we use a slight extension of the notation presented in Sect. 3 consisting in letting transitions be labeled by a *sequence* of output actions instead of a single destination!event pair. Such extension does not affect the semantics at a conceptual level, but helps very much in writing concise and effective specifications.

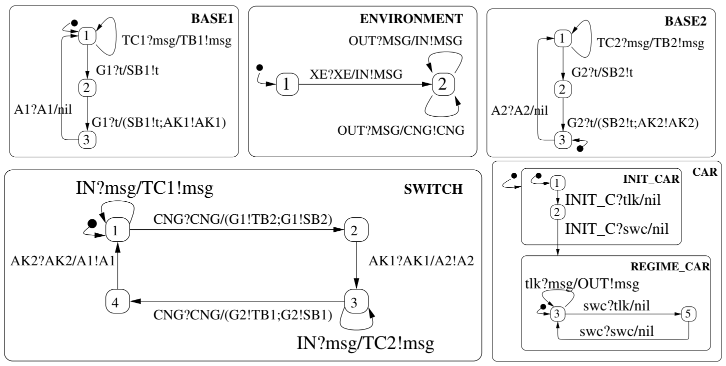


Fig. 4. The Hand-over Protocol $\delta\mu$Chart.

except INIT_C and XE, containing the values, <TB1:SB1> (TB1 at the top) and <XE> respectively. The ENVIRONMENT initially sends a message to the SWITCH via queue IN. Subsequently, on receiving a message from queue OUT, it may non-deterministically send the next message via queue IN or send the indication (CNG) that the car moved to the other cell, via queue CNG. Notice that activation of ENVIRONMENT is done via auxiliary queue/event XE. The switching center, modeled by SWITCH, in its initial state (1) is ready to receive either messages from queue IN or the information that the base station to which the car is connected must be changed (message CNG from queue CNG). Upon receiving a CNG indication in state (1) the SWITCH sends to BASE1 via queue G1 the data (TB2) and control (SB2) channels of BASE2, moving to state (2). Upon receiving a message from queue IN in state (1) the SWITCH forwards the message, bound to variable *msg*, to BASE1, via queue TC1, moving back to state (1). This message-forwarding loop goes on until/unless a CNG event arrives. In state (2) SWITCH waits for the ack AK1 from BASE1 via queue AK1, and after that it sends the wakeup event A2 via queue A2 to BASE2 and moves to state (3), which is symmetric to state (1). State (4) is symmetric to (2) with BASE1 and BASE2 swapped. The specification of the base stations, BASE1 and BASE2 should be self-explanatory. Also the specification of the CAR should be easy to understand. Notice that the current data channel is stored in variable *tlk* while the current control channel is kept in *swc*.

5 Conclusions

In this paper we presented an extension of μCharts as a first step towards modeling mobility issues. In particular we addressed issues concerning device mobility, which imply the ability to dynamically change the system interconnection structure, by “opening” and “closing” connections (mobile computing [1]).

There are important features of mobile systems that cannot be expressed easily using only an asymmetric style of communication. Thus, our extension consists in letting the *trigger event* specification of transition labels be equipped with the explicit reference to the queue from which the event should be taken. Such a reference can also be a queue name variable, thus allowing dynamicity in the choice. A formal semantics definition has been given for the extension using deductive techniques, and some correctness results concerning requirements put by the official UML semantics have been shown. As an example of use of the new notation a specification of the Hand-over protocol has been provided.

It is worth pointing out that our (core) semantics definition is a slight extension of the definition we proposed in [7]. In particular it preserves its hierarchical/recursive nature. We have used the latter definition (or minor extension thereof) for covering several aspects of UMLSCs, like stochastic ones, model-checking, and formal testing. This proves the hight modularity and flexibility of our approach. Moreover, the use of recursive definitions on hierarchical structures greatly simplified the proofs of the properties of interest.

We plan to define a mapping from $\delta\mu$Charts to PROMELA for efficient model-checking with SPIN [5]. Such work will be based on similar work we have already successfully done for μCharts [6]. The PROMELA translator for μCharts is currently being implemented by Intecs Sistemi s.r.l. in the context of the project PRIDE funded by the Italian Space Agency. Another line of research which we are currently investigating is to further extend our model with localities in order to explicitly model dynamic code creation/removal and mobility as migration in the sense of *mobile computations* [1]. Finally, there are several features of UMLSCs which we did not address in the present paper but which can be added to our model. Object features are already dealt with in the above mentioned work on UML with localities and data/variables are already incorporated in the context of PRIDE. Exit/entry events can be dealt with as in [15]. Deferred events can be incorporated by extending the selection predicate *Sel*. History states can be modeled by a proper re-definition of function *DST* and the use of stores.

References

1. L. Cardelli and A. Gordon. Mobile ambients. In M. Nivat, editor, *FoSSaCS'98*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–145. Springer-Verlag, 1998.
2. P. Denning. Fault tolerant operating systems. *ACM Computing Surveys*, 8(4):359–389, 1976.
3. G. Ferrari, S. Gnesi, U. Montanari, M. Pistore, and G. Ristori. Verifying mobile processes in the HAL environment. In A. Hu and M. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
4. S. Gnesi, D. Latella, and M. Massink. Modular semantics for a UML Statechart Diagrams kernel and its extension to Multicharts and Branching Time Model Checking. *The Journal of Logic and Algebraic Programming. Elsevier Science*, 51(1):43–75, 2002.
5. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

6. D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing. The International Journal of Formal Methods. Springer-Verlag*, 11(6):637–664, 1999.
7. D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, pages 331–347. Kluwer Academic Publishers, 1999. ISBN 0-7923-8429-6.
8. D. Latella and M. Massink. Relating testing and conformance relations for UML Statechart Diagrams Behaviours. Technical Report CNUCE-B4-2002-001, Consiglio Nazionale delle Ricerche, Istituto CNUCE, 2002. (Full version).
9. D. Latella and M. Massink. On mobility extensions of UML Statecharts; a pragmatic approach. Technical Report 2003-TR-12, Consiglio Nazionale delle Ricerche, Istituto di Scienza e Tecnologie dell'Informazione 'A. Faedo', 2003.
10. E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In R. Shyamasundar and K. Euda, editors, *Third Asian Computing Science Conference. Advances in Computing Sience - ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 181–196. Springer-Verlag, 1997.
11. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992. Parts 1-2.
12. Object Management Group, Inc. *OMG Unified Modeling Language Specification - version 1.3*, 1999.
13. J. Philipps and P. Scholz. Compositional specification of embedded systems with statecharts. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice in Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 637–651. Springer-Verlag, 1997.
14. P. Scholz and D. Nazareth. Communication concepts for statecharts: A semantic foundation. In M. Bertran and T. Rus, editors, *Transformation-based Reactive Systems Development*, volume 1231 of *Lecture Notes in Computer Science*, pages 126–140. Springer-Verlag, 1997.
15. M. von der Beeck. A structured operational semantics for UML-statecharts. *Software Systems Modeling. Springer*, (1):130–141, 2002.

New Operators for the TURTLE Real-Time UML Profile

Christophe Lohr[1], Ludovic Apvrille[2],
Pierre de Saqui-Sannes[1,3], and Jean-Pierre Courtiat[1]

[1] LAAS-CNRS, 7 avenue du Colonel Roche,
31077 Toulouse Cedex 04, France
{lohr,courtiat}@laas.fr

[2] Concordia University, ECE department, 1455 de Maisonneuve W.,
Montreal, QC, H3G 1M8 Canada
apvrille@ece.concordia.ca

[3] ENSICA, 1 place Emile Blouin,
31056 Toulouse Cedex 05, France
desaqui@ensica.fr

Abstract. In a previous paper, we defined TURTLE, a Timed UML and RT-LOTOS Environment which includes a real-time UML profile with a formal semantics given in terms of translation to RT-LOTOS, and a model validation approach based on the RTL toolset. This paper presents an enhanced TURTLE with new composition operators (*Invocation*, *Periodic*, *Suspend / Resume*) and suspendable temporal operators which makes it possible to model scheduling constraints of real-time systems. The proposed extension is formalized in terms of translation to native TURTLE. Thus, we preserve the possibility to use RTL to check a real-time system model against logical and timing errors. A case study illustrates the use of the new operators.

1 Introduction

The Unified Modeling Language provides a general purpose notation for software system description and documentation. The fathers of the OMG standard [7] expected UML to evolve and to be specialized for specific classes of systems. Therefore, they introduced the concept of profile[1] to specialize the UML meta-model into a specific meta-model dedicated to a given application domain [10].

Further, several lobbyists at OMG have worked on a profile for real-time systems, published in early 2002 [8] and on the next version of UML, UML 2.0, an enhanced UML with real-time oriented features [12]. Although UML 2.0 opens promising avenues for real-time system modeling, the draft document [12] does not provide enhanced UML with a methodology or a formal semantics. Also, the expression power

[1] A profile may contain selected elements of the reference meta-model, extension mechanisms, a description of the profile semantics, additional notations, and rules for model translation, validation, and presentation.

of UML 2.0 in terms of real-time constraints modeling remains limited and compares to the one offered by SDL language [11].

UML's limitations in terms of real-time system design [10] have also stimulated research work on joint use of UML and formal methods. In [1], we defined TURTLE (*Timed UML and RT-LOTOS Environment)*, a real-time UML profile with a formal semantics expressed in RT-LOTOS [4] and a model validation tool (RTL [9]). TURTLE addresses the structure and behavioral description of real-time systems. A TURTLE class diagram structures a system in so-called "Tclasses". A *Tclass* behavior is described with an activity diagram enhanced with synchronization and temporal operators *à la* RT-LOTOS. Also, unlike UML 1.4, associations between Tclasses are not used for documentation purpose, but are attributed with a composition operator that gives them a precise semantics. The core of what we call "native TURTLE" includes four composition operators (*Sequence*, *Parallel* and *Synchro*, and *Preemption)*. If these four operators enable description of various systems [2] [5], their application to complex mechanisms such as task scheduling remains fastidious. The designer's burden is important since he or she must describe the details of mechanisms that might advantageously be modeled by high-level and "compact" operators. Suspending tasks is an example of such mechanisms commonly used in real-time systems.

The purpose of this paper is to define new high-level operators whose semantics is given in terms of native TURTLE: *Invocation*, *Periodic and Suspend / Resume*,. *Invocation* can be compared to a method call, but applies to inter-task and rendezvous communications through so-called "gates." *Periodic* and *Suspend / Resume* are used to model task scheduling.

The paper is organized as follows. Section 2 presents native TURTLE. Sections 3, 4 and 5 introduce the *Invocation*, *Periodic*, and *Suspend / Resume* operators, respectively. Section 6 illustrates the use of these operators. Section 7 concludes the paper.

2 The TURTLE Profile

2.1 Methodology

The purpose of the TURTLE profile is not to support the entire development of real-time systems. The goal remains "a priori validation" or, in other words, the possibility for a system designer to check a model against errors before the coding phase begins. Fig. 1 depicts an incremental life cycle where a priori validation follows the design and specification phases. A system specification is expressed as a combination of use-case and sequence diagrams. The purpose of the design phase is to produce a class diagram describing the structure of the system under design. The internal behaviors of active classes are modeled by means of activity diagrams. At last, class activity diagrams are combined together and then translated to a RT-LOTOS specification using the algorithms published in [5]. The so-obtained RT-LOTOS specification can be validated using RTL [4]. Also, a correspondence table between actions on the reachability graph and the TURTLE actions is built during the translation process [5]. Thus, system analysis may be done without any knowledge of RT-LOTOS.

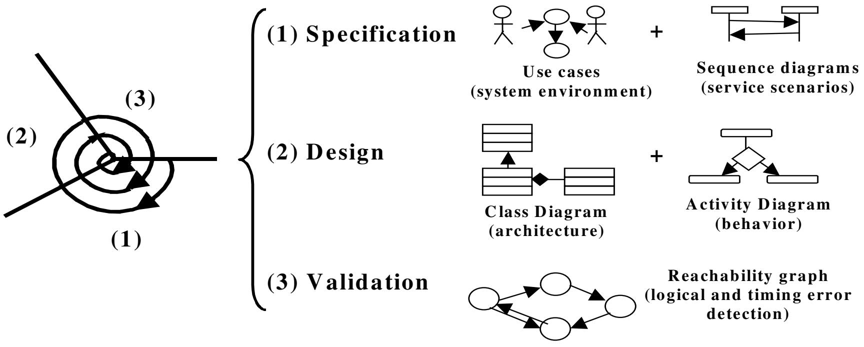

(1) Specification
(2) Design
(3) Validation
(1)
(2)
(3)
Use cases (system environment)
Sequence diagrams (service scenarios)
Class Diagram (architecture)
Activity Diagram (behavior)
Reachability graph (logical and timing error detection)

Fig. 1. TURTLE-based life cycle

2.2 Extended Class Diagram

A TURTLE class diagram is made of "normal" classes and stereotyped ones that we call *Tclass*. A *Tclass* is pictorially identified by a "turtle" icon located in the upper right corner of the *Tclass* symbol. Two *Tclasses* can synchronize with each other and exchange data using so-called "gates" (Fig. 2a). TURTLE introduces the *InGate* and *OutGate* abstract types to distinguish between input and output gates, respectively. The profile enables precise description of synchronization and parallelism relations between *Tclasses*. An association between two *Tclasses* must be attributed with the associative class of a composition operator. Operators available with native TURTLE are *Sequence, Parallel* (Fig. 2b), *Synchro*, and *Preemption*. The first three operators make it possible to express tasks that execute in sequence, tasks that run in parallel without communicating and task that synchronize[2] on gates. *Preemption* gives a task the possibility to interrupt another task forever.

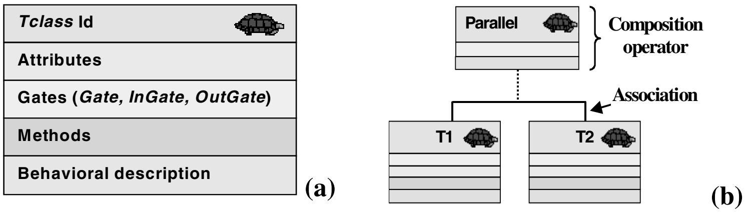
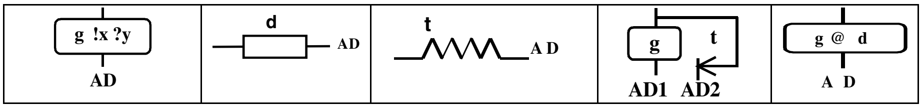


Fig. 2. Structure of a Tclass (a) and composition between two Tclass (b)

TURTLE extends UML activity diagrams to describe synchronized data sending and receiving, fixed (deterministic) delay, variable (non-deterministic) delay, time-limited offer, and time capture. Fig. 3 depicts the symbols associated with these operators (AD stands for Activity Diagram, the activity sub-diagram following the considered operator). The @ operator stores the amount of time elapsed between the offer of an action and its execution.

[2] *Synchro* must be accompanied by an OCL formula to express "connections" between the gates involved in the synchronization (*E.g.*{T1.g1 = T2.g2}).



Synchronization on g	Deterministic delay	Non-deterministic delay	Time-limited offer	Time capture

Fig. 3. New symbols for activity diagrams

3 *Invocation* Operator

3.1 Principle

Method call is a fundamental feature of object-oriented languages and of UML class diagrams in particular. With native TURTLE, the modeling of method call requires the use of two *Synchro* operators. Moreover, the activity diagram of the two *Tclasses* involved in the two synchronizations must check for the validity of data exchange performed during that synchronization. For example, when returning from a method call, data should be exchanged only from the callee to the caller. Such complexity leads us to introduce a novel operator –called Invocation – which makes it possible for a *Tclass* to insert the activity of another *Tclass* in its execution flows. *Invocation* differs from *Synchro* since the latter characterizes synchronization between two separate execution flows.

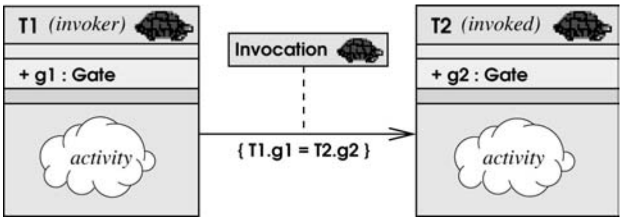


Fig. 4. Association attributed with an *Invocation* operator

Fig. 4 depicts an *Invocation* associative class. The class is attached to an association starting at *Tclass* T1 (*Invoker*) to heading to *Tclass* T2 (*Invoked*). Then, we say that T2 can be invoked by T1. Like synchronization, an invocation involves one gate in each class it concerns. Let us call g1 and g2 two gates ofT1 and T2, respectively. Let us assume the OCL formula *{T1.g1 = T2.g2}* is attached to the relation. Then, when T1 performs a call on g1, T1 must wait for T2 to perform a call on g2. When T2 performs the expected call, data can only be exchanged from T1 to T2, following the direction indicated by the arrow. In other words, parameters can be passed from T1 to T2. T1 is blocked until T2 performs a new call on g2. Call return values and other data can be passed from T2 to T1. Finally, controlled data exchange makes the *Invocation* operator far more complex that the use of two basic synchronization operators.

3.2 Example

Fig. 5 depicts a browsing system model. It contains an *Invocation* between a *BrowserWindow* and a communication network *Comm*, playing *Invoker* and *Invoked* roles, respectively. The values passed by *Invoker* are prefixed by "!". The return value expected by *Invoker* is prefixed by "?".

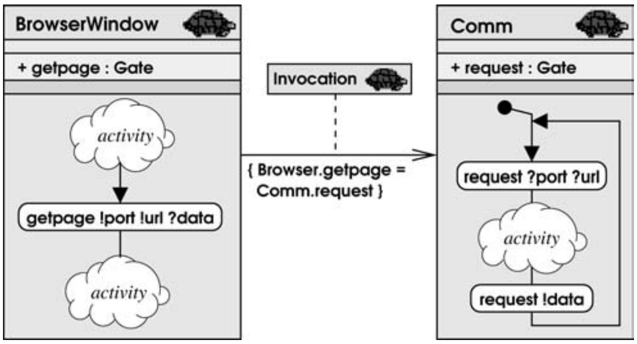


Fig. 5. Use of an *Invocation* operator in a browsing system

3.3 Translation to Native TURTLE

Let us consider a TURTLE modeling whose class diagram contains an *Invocation* operator. The translation of this modeling to native TURTLE requires modifications at class and activity diagrams level (Fig. 6).

Class Diagram. The *Invocation* operator is substituted with a *Synchro* operator which uses the same synchronization gate as the *Invocation* operator.

Activity Diagram. Let us consider Invoker, the *Tclass* at the origin of the navigation arrow. The call on the gate related with *Invocation* is replaced by two synchronization calls. First synchronization includes a data offer ("!" prefix) which corresponds to a parameter passing in a method call. Second synchronization consists in waiting for return values (variable prefixed by "?"). The diagram of the *Invoked* class is not modified.

4 Periodic

4.1 Principle

The native TURTLE profile lacks a high-level operator for the description of periodic tasks. Usually, a periodic task executes its main activity in a timely and regular basis. But we propose to introduce a more powerful *Periodic* composition operator. Indeed, *Periodic* makes it possible to specify that a *Tclass* T2 starts *t* units of time after another *Tclass* T1 has started. Also, a *Periodic* operator attached to an association linking a *Tclass* to itself describes a "periodic class" *i.e.* a class whose main activity is periodically started.

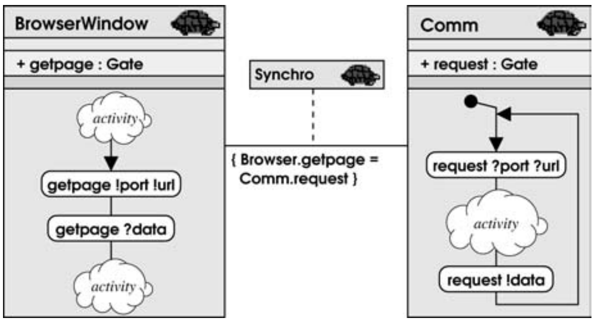


Fig. 6. Translating an *Invocation* into native TURTLE

4.2 Example

Let us consider a *Periodic* associative class attached to an association directed from a *Tclass* T1 to a *Tclass* T2 (Fig. 7). T2 is activated 100 units of time after T1's completion (*cf.* the keyword "period" in the OCL formula). The keyword "deadline" specifies a maximal execution time for T2. Note that "deadline" is optional. Also, T3 is a periodic class *i.e.* its activity is started every 100 time units and each activity must have completed within 50 time units.

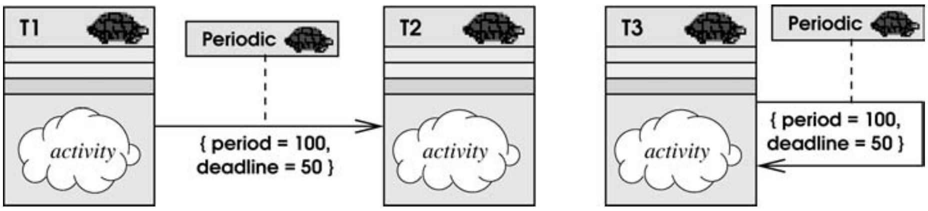

T1
activity
Periodic
{ period = 100, deadline = 50 }
T2
activity
T3
activity
Periodic
{ period = 100, deadline = 50 }

Fig. 7. Association attributed by a *Periodic* operator

4.3 Translation to Native TURTLE

Be two *Tclasses* T1 and T2 linked by an association attributed with a *Periodic* operator. This association is characterized by a navigation arrow from T1 to T2. The proposed translation algorithm inserts a new *Tclass* named *Deferment* between T1 and T2, and two composition operators: one *Parallel* operator between T1 and *Deferment* and one *Sequence* operator between *Deferment* and T2. The value of the delay operator in *Deferment*'s activity diagram is equal to the value of the "period" keyword in the original diagram. Also, T2 execution time should not exceed the "deadline" value if the latter is specified. If so, a non-intrusive observer is added to the diagram (see the *Tclass Observer* on Fig. 8). Thus, if T2's execution time goes beyond the value specified by "deadline" keyword, then the observer preempts T2 and performs a synchronization on gate *error*. The occurrence of that synchronization is easy to point out at verification step. Indeed, "*error*" appears as a transition label in the reachability graph of the RT-LOTOS code derived from the TURTLE model.

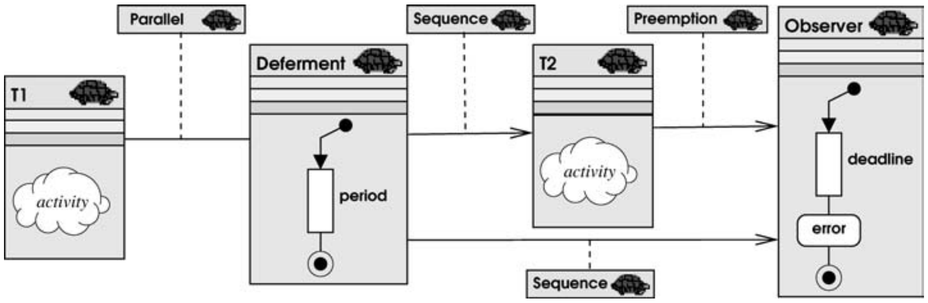


Fig. 8. Translating the *Periodic* operator into native TURTLE

5 Suspend/Resume

5.1 Principle

The native TURTLE profile defined in [1] includes a *Preemption* operator which allows a *Tclass* T1 to abruptly interrupt a *Tclass* T2 without any possibility for T2 to keep track of its activity and resume later on. Thus, TURTLE lacks a high-level operator to express the possibility for a *Tclass* to be suspended and subsequently resumed. The *Suspend*[3] operator introduced in this section answers that need.

Fig. 9 depicts a *Suspend* associative class attached to an association directed from a *Tclass* T1 (*Suspender)* to a *Tclass* T2 (*Suspended*). T2 can be suspended and reactivated by T1. Both operations require a call on gate *s* (*s* appears in the OCL formula associated with the relation from T1 to T2). When T1 performs a call on *s*, T2's activity is suspended. Then, the next call on s performed by T1 resumes T2. T1 can suspend and resume T2 as many times as needed.

5.2 Example

TaskManager (see Fig. 10) implements a basic scheduler which switches *TaskA* and *TaskB*, two tasks sharing a same processor resource. Indeed, *TaskManager* can suspend (or resume) *TaskA* and *TaskB* using gates *SwitchA* and *SwitchB*, respectively.

[3] *Suspend* is an abbreviation for *Suspend/Resume*.

When the application starts, both tasks are in the active state. But they are immediately suspended by *TaskManager* (*cf.* the calls on *SwitchA* and *SwitchB* at the beginning of *TaskManager*'s activity diagram). Then, *TaskA* and *TaskB* are activated one after the other during a quantum of time. The attribute *quantumA* (resp. *quantumB*) denotes the quantum of time allocated to *TaskA* (resp. *TaskB*).

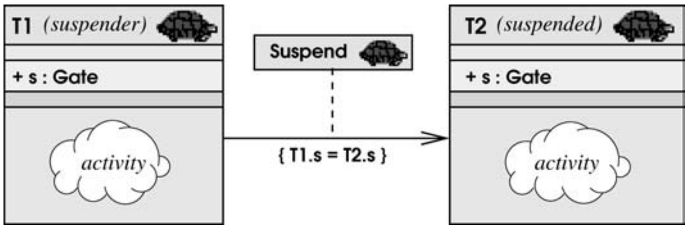


Fig. 9. Association attributed with a *Suspend* operator

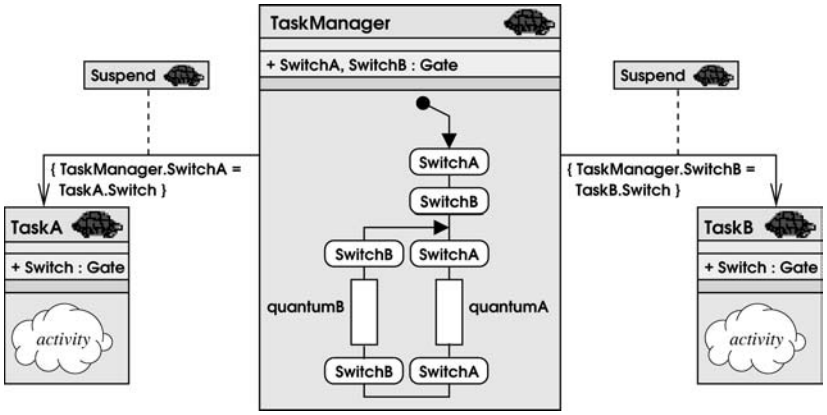


Fig. 10. Example of a basic scheduler modeled using the *Suspend* operator

5.3 Suspendable Temporal Operators

The native TURTLE temporal operators presented at Fig. 3 assume a universal time. Time continuously progresses and can't be suspended. Further, it uniformly applies to all components of the system under design. These temporal operators can be used to model, for example, a timer but also to model the execution time of an algorithm. But now, because of the introduction of the *Suspend* composition operator, a TURTLE *Tclass* execution might be interrupted. When interrupted, the timer of a class should continue to elapse. Conversely, a temporal operator modeling the execution time of an algorithm should be stopped when the class is interrupted. This leads us to introduce temporal operators that support the concept of time suspension and resume. We call them "suspendable temporal operators".

Fig. 11 depicts the three new suspendable temporal operators: a "suspendable" deterministic delay, a "suspendable" non-deterministic delay, and a "suspendable" timed-limited offer. A small hourglass is added to the original symbols to express the concept of time suspension.

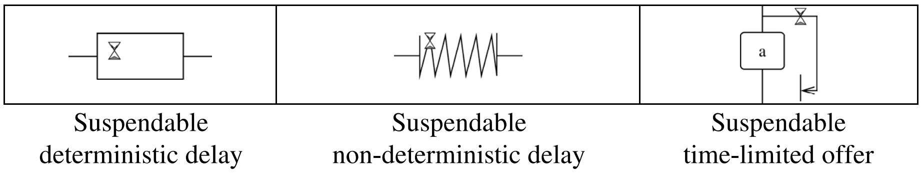

a
Suspendable deterministic delay
Suspendable non-deterministic delay
Suspendable time-limited offer

Fig. 11. Suspendable deterministic delay, non-deterministic delay, and time-limited offer

Fig. 12 illustrates the use of the deterministic (rectangle box) and non-deterministic (spring) delay operators in their original and "suspendable" versions. The system under design is a DVD player for Personal Computer. First, the DVD player takes a disk and then, it checks it. The access time to tracks depends on the mechanical properties of the device. Mechanical operations cannot be interrupted. Therefore, they are completed independently of any other computations on the PC. Conversely, *Task-Manager* can suspend data decoding which follows synchronization on *getData*. As a consequence, we associate an hourglass with the deterministic and non-deterministic delays on the left branch of *DVD_Player*'s activity diagram.

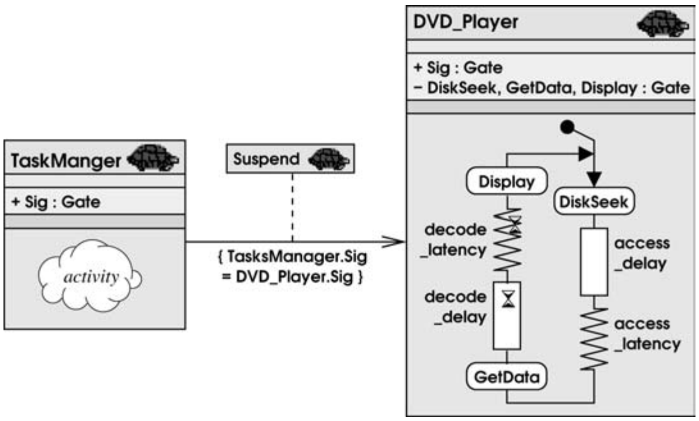


Fig. 12. Example of using basic and suspendable delays

5.4 Translation to Native TURTLE

In order to manage the *Suspend* composition operator, the system should take into account:

- The state of all objects that can be suspended. The state is either *active* or *suspended*.
- The actions that were currently being executed when the suspension was invoked. In particular, the amount of time already consumed by temporal operators should be memorized when a suspension occurs

Detecting the state of an object can be done as follows. If *s* denotes the gate used by a *Tclass* T1 to suspend a *Tclass* T2, the state of T2 can be computed from the number of calls on gate *s*. The first call on *s* suspends T2. The second one resumes T2. And so on. In practice, the state of an object can be obtained from the parity of the number of calls performed on *s*: if the parity is odd, the object is in *active* state. Otherwise, it is in *suspended* state.

As a consequence, we propose to translate class and activity diagrams as follows. A Synchro operator takes the place of each *Suspend* operator. The OCL formula of *Suspend* is not modified because *Synchro* operates on the same gate *s*. An association attributed by *Synchro* is also created between the *Suspended Tclass* T2 and a new *Tclass* named *ParitySig*. T2 and *ParitySig* synchronize with each other on gates *quest* and *s* (Fig. 13). *ParitySig* checks the parity *p* of the number of occurrences of action *s*. Parameter *p* is odd when T2 is suspended and is even when T2 is active. T2 uses *quest* to question *ParitySig* about *p* (a boolean value exchanged at synchronization time on *quest).*

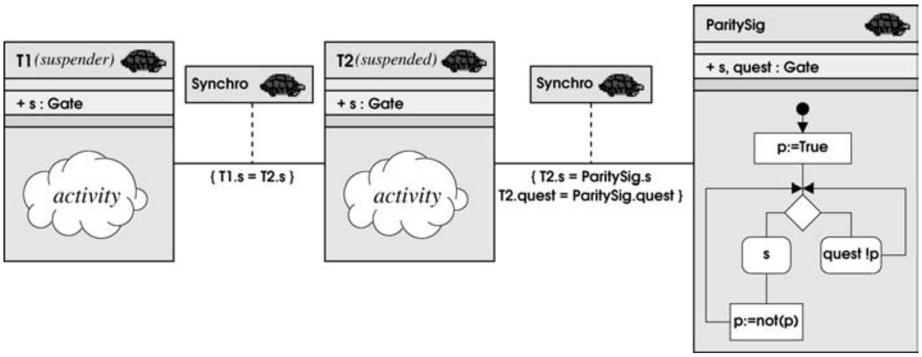


Fig. 13. Translation of the *Suspend* composition operator

T2's activity diagram is modified as follows. The sub-diagram depicted on Fig. 14 is inserted before each "suspendable" operator symbol. This sub-diagram checks the parity of *p*. If *p* is true (even parity) then T2's activity goes on. Otherwise, T2 waits on gate *s* for a resume action.

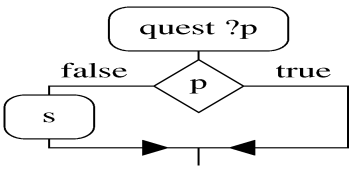


Fig. 14. Testing interruption parity

A second problem is to memorize the action currently executed by T2 when the suspension occurred. A translation scheme is now proposed for every action that needs to be memorized in suspended state.

- *Synchronization on gates*. The synchronization action on a gate *a* (left part of Fig. 15) is replaced by a sub-diagram involving gates *a* and *s* (suspension gate, see right part of Fig.15). On this diagram, the empty diamond represents a non-deterministic choice between gates a and s. When T2 is suspended by a synchronization action on s, it must wait for a resume action modeled by a second synchronization on s.

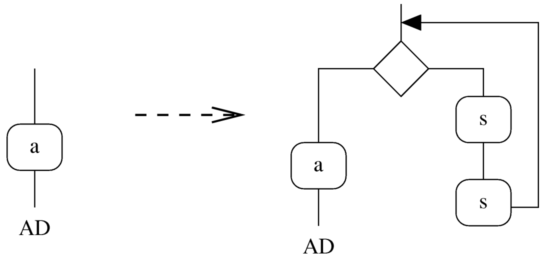


Fig. 15. Translation of gates for suspendable *Tclasses*

- *Suspendable Temporal intervals.* The left part of Fig. 16 depicts a deterministic delay of value dmin, followed by a non-deterministic delay of value dmax-dmin. The two operators in sequence represent a time interval equals to [dmin, dmax]. The temporal interval on the left part of Fig. 16 is translated to the sub-diagram on the right part of Fig. 16. The diamond models a non-deterministic choice between a "non-suspendable" temporal interval and a synchronization action on s. This first synchronization on s suspends T2's activity. The synchronization on s is limited to DMAX time units. The @ operator stores in variable d the time elapsed since the synchronization on s was offered. When the second synchronization action on s is performed, the new values of DMIN and DMAX are computed: DMIN and DMAX are decremented by d, where d denotes the time consumed within the temporal interval before suspension; and T2 resumes its execution (recursive call).

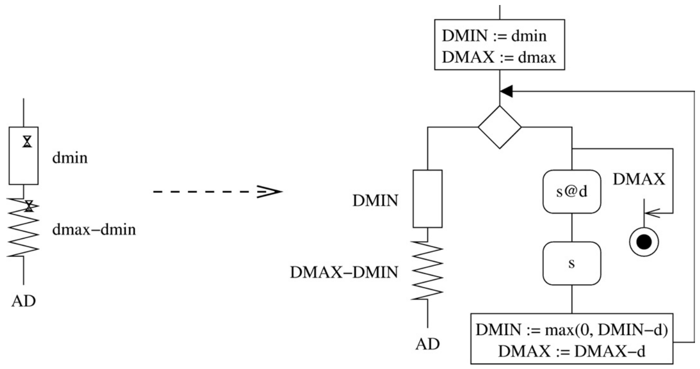


Fig. 16. Translation of suspendable time intervals

- *Time-limited offer.* A suspendable time-limited offer is depicted on the left part of Fig. 17. If no synchronization on gate a (left branch) occurs before L units of time, then, the sub-diagram AD2 is executed. Otherwise, AD1 is executed. The translation of the suspendable time-limited offer is depicted on the right part of Fig. 17: a choice is introduced between a "non-suspendable" time-limited offer on gate a (time limit of L) and a synchronization on action s. This first synchronization on s suspends T2's activity. The @ operator stores in variable d the time at which the suspension occurred (first synchronization on s). When a second synchronization on s is performed, the value of L is computed. Then, T2 resumes its execution (recursive call).

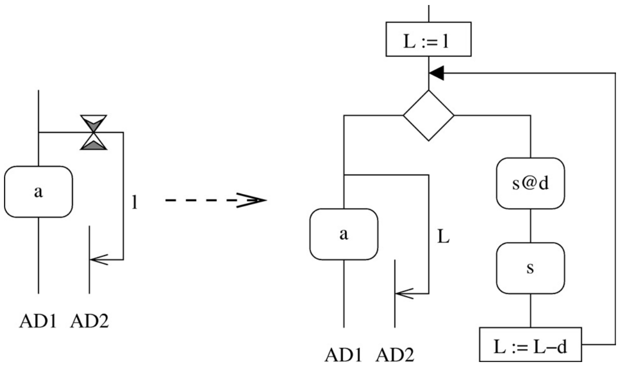


Fig. 17. Translating a suspendable time-limited offer

Note. To translate suspendable temporal operators, we use the @ RT-LOTOS operator to save the time at which suspension and resume actions occur. But the verification algorithms of RTL, the RT-LOTOS toolset, don't yet support the @ operator. As a consequence, only intensive simulations can be performed from TURTLE models containing *Suspend* composition operators.

6 Case Study

We propose to reuse a case study presented in [6]. The system under design is a protocol entity composed of *Sender* and *Receiver*. The purpose of the design depicted on Fig. 18 to model the communication functions of a user terminal and not to validate an end-to-end communication architecture. Therefore, we assume that *Sender* and *Receiver* run on the same processing unit.

Sender is submitted to the following requirements:

1) The sender must accept a request message from the user every ts time units.
2) The sender must process each message within t ∈ [es, Es] time units before sending it.

3) The sender must start processing the message by us time units after the user requested it (us < ts).
4) A message must be sent before xs time units after the request (Es < xs ≤ ts).
5) Receiver is constrained by similar requirements for processing, delivering and accepting received messages:
6) Two received messages are separated by at least tr time units.
7) Processing a received message takes t ∈ [er, Er] time units.
8) A message must be accepted before ur time units since its arrival (ur < tr).
9) A message must be processed and delivered before "xr" time units after its arrival (Er < xr ≤ tr).

Fig. 18 depicts three software components implemented on the same processing unit, namely Sender, Scheduler and Receiver. Receiver's activity diagram is built upon the same principle as Sender's. Receiver is not described for space reasons. Scheduler implements the following resource allocation politics. A receive request has priority over a sending request. When the processor is allocated to Sender, Scheduler is still ready to give the control to Receiver. Thus, if a receive request arrives, Sender is suspended and Receiver can execute till completion. Scheduler synchronizes on gate send in order to wait for Sender to complete its ongoing activity. Requirements 2, 3 and 4 and their counterparts in 6, 7 and 8 are implemented by three parallel sub-activities within Sender's activity (and similarly within Receiver's activity). Each parallel branch contains an i error action which represents a violation of the i^{th} requirement. Requirements 2, 3, 4 and 6, 7, 8 correspond to constraints related to computation time. Consequently, these constraints are modeled by "suspendable" temporal operators. Conversely, requirements 1 and 5 are not implemented by any processing subject to suspension. Indeed, their implementation use the deterministic delay operator defined in Section 2. Requirements 1 and 5 are checked against violation using the deadline value associated with the Periodic operator (using an observer, as depicted on Fig. 8).

Using the TURTLE extensions discussed in this paper dramatically reduces the size of class and activity diagrams. The size of *Sender*'s activity diagram is divided by four and the size of the class diagram is divided by two in terms of class and relation number. Clearly, the proposed high-level operators make the use of TURTLE more comfortable for complex real-time-system design.

7 Conclusions

In [1], we defined TURTLE, a real-time UML profile which improves the OMG-based notation in terms of real-time system modeling, formal semantics, and formal verification. The profile was successfully applied to various case studies, including an industrial application in the framework of the dynamic reconfiguration of software embedded on board satellites [2].

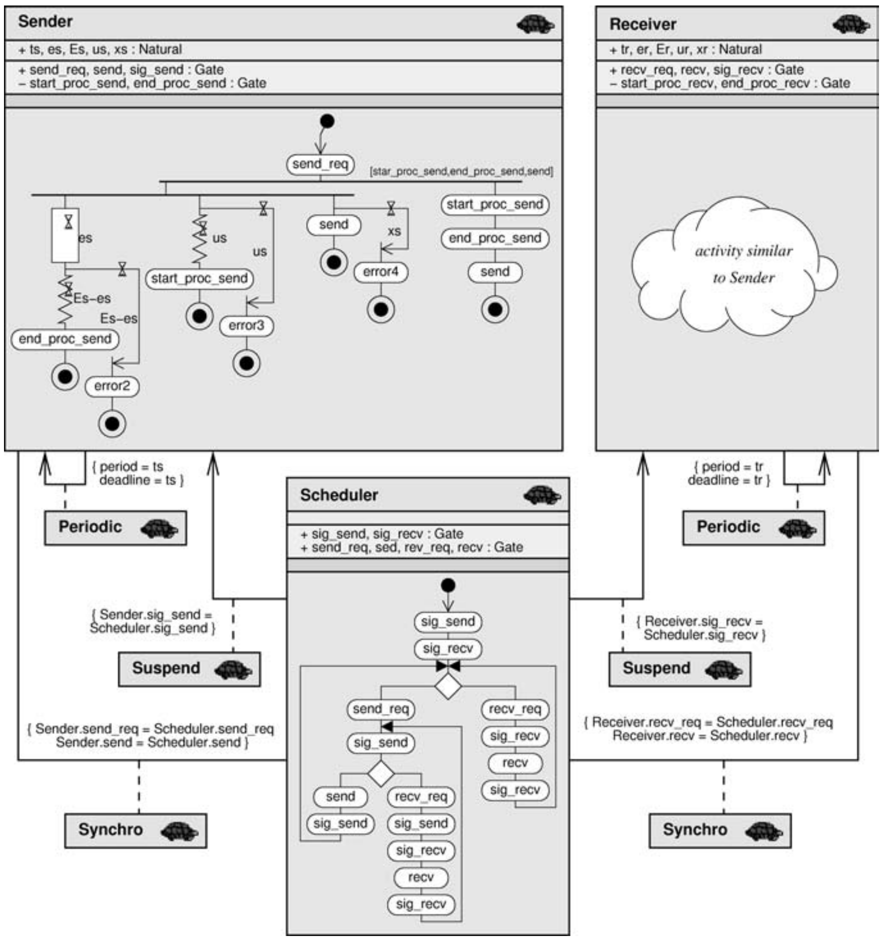

Sender
+ ts, es, Es, us, xs : Natural
+ send_req, send, sig_send : Gate
– start_proc_send, end_proc_send : Gate
send_req
[star_proc_send,end_proc_send,send]
es
us
us
send
xs
start_proc_send
end_proc_send
send
Es–es
Es–es
start_proc_send
error4
end_proc_send
error3
error2
Receiver
+ tr, er, Er, ur, xr : Natural
+ recv_req, recv, sig_recv : Gate
– start_proc_recv, end_proc_recv : Gate
activity similar to Sender
{ period = ts deadline = ts }
Periodic
{ period = tr deadline = tr }
Periodic
Scheduler
+ sig_send, sig_recv : Gate
+ send_req, sed, rev_req, recv : Gate
{ Sender.sig_send = Scheduler.sig_send }
Suspend
{ Receiver.sig_recv = Scheduler.sig_recv }
Suspend
sig_send
sig_recv
send_req
recv_req
sig_send
sig_recv
recv
sig_recv
send
recv_req
sig_send
sig_send
sig_recv
recv
sig_recv
{ Sender.send_req = Scheduler.send_req Sender.send = Scheduler.send }
{ Receiver.recv_req = Scheduler.recv_req Receiver.recv = Scheduler.recv }
Synchro
Synchro

Fig. 18. TURTLE class and activity diagrams of the case study

Despite of its expressiveness, TURTLE was missing high-level operators to model the scheduling of tasks in real-time systems. To address this issue, this paper enhances the TURTLE profile with three composition operators named *Invocation*, *Periodic*, and *Suspend/Resume*. *Suspend/Resume* leads to introduce the concept of "suspendable" temporal operators in activity diagrams. The proposed extensions are formalized in terms of translation to native TURTLE. This preserves the possibility to use RTL [9] to check a real-time system model against logical and timing errors. As shown in the case study, the use of new these new reduces TURTLE designs complexity. Also, powerful abstractions offered by such operators make them well suited for the modeling and validation of large-scale systems. TURTLE extends class and activity diagrams of UML 1.4. In [3], we defined TURTLE-P, a profile that enhances TURTLE with component and deployment diagrams. TURTLE-P better meets the needs of protocol and distributed architecture modeling and validation because it makes it possible to explicitly model the various execution sites of applications, and communication constraints between theses sites. Again, the TURTLE-P semantics is given in terms of RT-LOTOS.

References

1. Apvrille, L., de Saqui-Sannes, P., Lohr, C., Sénac, P., Courtiat, J.-P.: A New UML Profile for Real-time System Formal Design and Validation in Proceedings of the Fourth International Conference on the Unified Modeling Language (UML'2001), Toronto, Canada, October 1 – 5, 2001.
2. Apvrille L.: Contribution to Dynamic Reconfiguration of Embedded Real-Time Software: Application to a Satellite Telecommunication Environment, Ph.D. dissertation (in French), June 2002.
3. Apvrille L, de Saqui-Sannes, P., Khendek, F.: TURTLE-P : a UML Profile for Distributed architecture Validation (in French). Submitted for publication.
4. Courtiat, J.-P., Santos, C.A.S., Lohr, C., Outtaj, B.: Experience with RT-LOTOS, a Temporal Extension of the LOTOS Formal Description Technique, Computer Communications, Vol. 23, No. 12. (2000) p. 1104-1123.
5. Lohr, C.: Contribution to Real-Time System Specification Relying on the Formal Description Technique RT-LOTOS. Ph.D. dissertation (in French), December 2002.
6. Hernalsteen, C.: Specification, Validation and Verification of Real-Time Systems in ET-LOTOS. Ph.D. thesis, Université Libre de Bruxelles, Belgium (1998).
7. Object management Group : Unified Modeling Language Specification, Version 1.4, http://www.omg.org/cgi-bin/doc?formal/01-09-67, 2001.
8. Object Management Group : UML Profile for Scheduling, Performance, and Time, Draft Specification, ftp://ftp.omg.org/pub/docs/ptc/02-03-02.pdf .
9. Real-time LOTOS. http://www.laas.fr/RT-LOTOS .
10. Terrier, F., Gérard, S., Real Time System Modeling with UML: Current Status and Some Prospects, Proceedings of the 2nd Workshop of the SDL Forum society on SDL and MSC, SAM 2000, Grenoble, France, 2000.
11. Tau Generation 2. http://www.telelogic.com/products/tau/index3.cfm .
12. UML 2.0. http://www.u2-partners.org Baldonado, M., Chang, C.-C.K., Gravano, L., Paepcke, A.: The Stanford Digital Library Metadata Architecture. Int. J. Digit. Libr. 1 (1997) 108–121.

Checking Consistency in UML Diagrams: Classes and State Machines[⋆]

Holger Rasch and Heike Wehrheim

Universität Oldenburg, Department Informatik
26111 Oldenburg, Germany
{rasch,wehrheim}@informatik.uni-oldenburg.de

Abstract. One of the main advantages of the UML is its possibility to model different views on a system using a range of diagram types. The various diagrams can be used to specify different aspects, and their combination makes up the complete system description. This does, however, always pose the question of *consistency*: it may very well be the case that the designer has specified contradictory requirements which can never be fulfilled together.
In this paper, we study consistency problems arising between static and dynamic diagrams, in particular between a class and its associated state machine. By means of a simple case study we discuss several definitions of consistency which are based on a common formal semantics for both classes and state machines. We furthermore show how consistency checks can be automatically carried out by a model checker. Finally, we examine which of the consistency definitions are preserved under refinement.

1 Introduction

The UML (Unified Modeling Language) [21] is an industrially accepted standard for object-oriented modelling of large, complex systems. The UML being a unification of a number of modelling languages offers various diagram types for system design. The diagrams can roughly be divided into ones describing *static* aspects of a system (classes and their relationships) and those describing *dynamic* aspects (sequences of interactions). For instance, class diagrams fall into the first, state machines and sequence diagrams into the second category. While in general it is advantageous to have these different modelling facilities at hand, this also poses some non-trivial questions on designs. The different views on a system as described by different diagrams are not orthogonal, and may thus in principle be inconsistent when combined.

While it is an accepted fact that consistency is an issue in UML-based system development, appropriate definitions of consistency are still an open research topic. In this paper we propose and discuss definitions of consistency between static and dynamic diagrams, more precisely, between a class and its associated state machine. We aim at a *formal definition* of consistency, and thus will first

[⋆] This research is partially supported by the DFG under grant OL 98/3-1.

give a formal semantics to both class definitions and state machines. This in particular requires the use of a formal specification language for classes to precisely fix the types of attributes and the semantics of methods. Since the UML does not prescribe a fixed syntax for attributes and methods of classes we feel free to choose one. Here, we have chosen Object-Z [19], an object-oriented specification language appropriate for describing static aspects of systems.

Since we aim at a formal definition of consistency we need a *common* semantic domain for classes and state machines in which we can formulate consistency. This common semantic domain is the failures-divergences model of the process algebra CSP [14, 18]. This choice has a number of advantages: on the one hand a CSP semantics for Object-Z is already available [9, 19, 20], on the other hand CSP has a well developed theoretical background as well as tool-support in the form of the FDR model checker [12]. For (restricted classes of) state machines a CSP semantics has already been given in [5]; we give another one for a simple form of *protocol state machines* in this paper. The first step during a consistency check is always the translation of class description and state machine into CSP.

The translation gives us the CSP descriptions of two different views on a class: one view describing attributes of classes and the possible effects of method execution (data dependent restrictions) and another view describing allowed orderings of method executions. These two descriptions are the basis for several consistency definitions. The property of consistency should guarantee that the two restrictions imposed on the behaviour of (an instance of) a class are not completely contradictory. There might, however, be different opinions as to what this means, ranging from "there is at least one possible run of the model" to "every method should always eventually be offered to the environment". The various forms of consistency are formally specified, and on the case study it is discussed what the effects of requiring such forms of consistency are. Our second focus is on *tool-supported consistency checks*: for each definition we show how the FDR model checker can be employed to automatically carry out the check.

During the development process a model may gradually be altered towards one close to the actual implementation domain. When consistency has been shown for a model developed in earlier phases, successive transformations of the model should preserve consistency if possible. In a formal approach to system development *refinement* is most often used as a correctness criterion for model evolutions. The question is thus whether the proposed consistency definitions are preserved under refinement. In our case, there are two notions of refinement to be considered: *data refinement* in the state-based part and *process refinement* (viz. failures-divergences refinement in CSP) in the behaviour-oriented part. Fortunately, we can restrict our considerations to process refinement since data refinement in Object-Z is known to induce failures-divergences refinement on the CSP processes obtained after the translation [20, 15]. For each of the definitions we hence either prove preservation under refinement or illustrate by means of a counterexample that preservation cannot be achieved in general.

The paper is structured as follows. In the next section we introduce the small case study and give a brief introduction to Object-Z. Section 3 explains the translation of both the class and the state machine into CSP. The result of this translation is the basis for defining and discussing consistency in Sect. 4. Finally, Sect. 5 presents the results on preservation under refinement.

2 Case Study

The small case study which we use to illustrate consistency definitions concerns the modelling of an elevator class. It is a typical example of a class with a model that contains a static part (attributes and methods) as well a dynamic part which describes allowed orderings of method executions.

The static class diagram of the elevator specification shown in Figure 1 models the class with its attributes and methods.

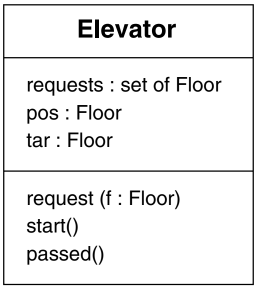


Fig. 1. Class Elevator

Its attributes are *requests* (to store the current requests for floors), *pos* (the current position of the elevator) and *tar* (the next target). It has a method *request* (to make requests for particular floors), a private method *start* (to start the elevator once there is a pending request) and a method *passed* which is invoked when the elevator is moving and has passed a certain floor.

The Object-Z specification below[1] gives a more precise description of this class. It formally specifies the types of the attributes and the semantics of methods. For each method we give a *guard* (an enabling schema) defining the states (i.e. valuations of attributes) of the class in which the method is executable and an *effect* defining the effects of method execution on values of attributes.

The specification starts with the definition of type *Floor*.

$$\begin{array}{|l}
minFloor, maxFloor : \mathbb{N}_1 \\
\hline
minFloor < maxFloor
\end{array}
\qquad\qquad Floor == minFloor..maxFloor$$

The class specification itself consists of an interface, a state schema, an initialisation schema and enable and effect schemas for methods. The interface consists of the method of the class itself (with keyword `method`) plus those called by the class (keyword `chan`). Input parameters of methods are marked with ?.

[1] To be more specific, it is the Object-Z part of a CSP-OZ specification [9].

When an enabling schema for a method is left out it corresponds to a guard which is always *true*. Effect schemas refer to the values of attributes after execution of a method by using primed versions of the attributes. The Δ-list of a method specifies the attributes which are changed by method execution.

Elevator

$\texttt{method}\ \mathit{request} : [f? : \mathit{Floor}]$
$\texttt{method}\ \mathit{passed}, \mathit{start}$
$\texttt{chan}\ \mathit{closeDoor}, \mathit{openDoor}$
$\texttt{chan}\ \mathit{stop}, \mathit{up}, \mathit{down}$

$\mathit{requests} : \mathbb{F}\,\mathit{Floor}$
$\mathit{pos}, \mathit{tar} : \mathit{Floor}$

Init
$\mathit{pos} = \mathit{tar} = \mathit{minFloor}$
$\mathit{requests} = \varnothing$

enable_*up*
$\mathit{pos} < \mathit{tar}$

enable_*down*
$\mathit{pos} > \mathit{tar}$

effect_*request*
$\Delta(\mathit{requests})$
$f? : \mathit{Floor}$

$f? \neq \mathit{pos} \Rightarrow$
$\quad \mathit{requests}' = \mathit{requests} \cup \{f?\}$
$f? = \mathit{pos} \Rightarrow \mathit{requests}' = \mathit{requests}$

enable_*stop*
$\mathit{pos} = \mathit{tar}$

enable_*start*
$\mathit{requests} \neq \varnothing$

enable_*passed*
$\mathit{pos} \neq \mathit{tar}$

effect_*start*
$\Delta(\mathit{tar}, \mathit{requests})$

$\mathit{tar}' \in \mathit{requests}$
$\mathit{requests}' = \mathit{requests} \setminus \{\mathit{tar}'\}$

effect_*passed*
$\Delta(\mathit{pos})$

$\mathit{pos} > \mathit{tar} \Rightarrow \mathit{pos}' = \mathit{pos} - 1$
$\mathit{pos} < \mathit{tar} \Rightarrow \mathit{pos}' = \mathit{pos} + 1$

enable_*openDoor*
$\mathit{pos} = \mathit{tar}$

enable_*closeDoor*
$\mathit{pos} \neq \mathit{tar}$

This is the static part of the model, specified by a class diagram. It fixes all data-dependent aspects of the class. Next, we model the dynamic view on an elevator. Figure 2 shows the state machine for class Elevator. This is an extended protocol state machine, which in addition to specifying the order for calls to the methods of the corresponding class also includes the methods *called* by (instances of) the class.

It consists of two submachines in parallel. The first submachine specifies the allowed sequences in the movements of the elevator. First, the elevator starts (this means picking a target from the available requests), the door is closed, and the elevator is send either up or down. During movement some floors are passed and eventually the elevator is stopped and the door opened again. Requests can be made at any time, thus the state machine specifying requests is concurrent to the movements state machine. This completes the model for class Elevator.

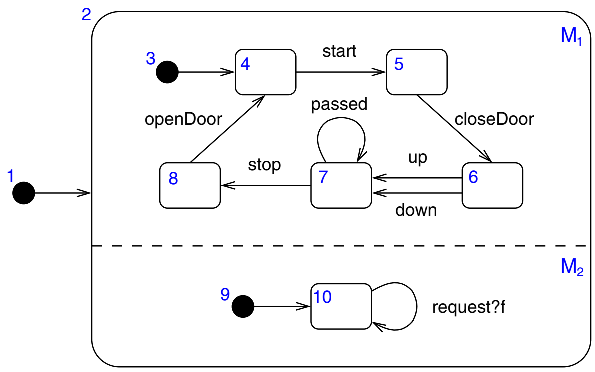


Fig. 2. Protocol of class Elevator

3 Translation into the Semantic Model

The first step in checking consistency of class definitions and state machines is their translation into a common semantic domain. The semantic domain we have chosen here is a semantic model of the process algebra CSP[2]. Instead of directly giving a (trace based) semantics to classes and state machines we translate them into CSP. This way the result remains readable and is furthermore amenable to checks with the FDR model checker for CSP.

3.1 Translation of the Object-Z Specification

The translation of class Elevator follows a general translation scheme for Object-Z developed in [11][3]. The basic idea is that a class is translated into a parameterised process. The parameters of the process correspond to the attributes of the class. For each method of the class (like *passed*) and each method called by the class (like *closeDoor*) a separate channel is used[4]. Parameters and return values are encoded as data sent on the channels.

Each method is translated to a recursion of the main process, guarded by the event prefix ($e \rightarrow$) corresponding to the method and possibly modifying the process parameters according to the effect schema of the method. All enabled methods are offered to the environment using external choice ($\Box$). Not offering the disabled methods, i.e. the translation of the enable schemas, is achieved by using guards (b &). Internal nondeterminism ($\sqcap$) possibly needed for updating the state space or choosing parameters for method calls is always 'below' the external choice and must not influence the set of offered events. Finally, the *Init* schema is translated by specifying the initial process parameters. Again, this

[2] CSP has several semantic models. For the general semantics the *failures-divergences* model is used, but some of the checks studied here use less powerful models.

[3] This is a partial map; some abstractions cannot be handled. It can be automated.

[4] Method calls are modelled as CSP communication.

requires a nondeterministic choice over all possible valuations, if there is more than one.

The result of this part of the translation is shown below. The attributes *pos*, *tar* and *requests* of the Object-Z are represented by the process parameters p, t and R, respectively[5]. The notation $T \lessdot b \gtrdot E$ is the operator notation of conditional choice meaning 'if b then T else E'.

$$
\begin{array}{rcl}
PROC_Z & = & Z(minFloor, minFloor, \varnothing) \\
Z(p,t,R) & = & \quad p = t \;\&\; \text{openDoor} \rightarrow Z(p,t,R) \\
& & \Box\; p \neq t \;\&\; \text{closeDoor} \rightarrow Z(p,t,R) \\
& & \Box\; p = t \;\&\; \text{stop} \rightarrow Z(p,t,R) \\
& & \Box\; \text{request?f} \rightarrow Z(p,t,R \cup \{f\} \lessdot f \neq p \gtrdot R) \\
& & \Box\; R \neq \varnothing \;\&\; \left(\bigsqcap_{t' \in R} \text{start} \rightarrow Z(p,t',R \setminus \{t'\})\right) \\
& & \Box\; p \neq t \;\&\; \text{passed} \rightarrow Z(p+1 \lessdot p < t \gtrdot p-1, t, R) \\
& & \Box\; p < t \;\&\; \text{up} \rightarrow Z(p,t,R) \\
& & \Box\; p > t \;\&\; \text{down} \rightarrow Z(p,t,R)
\end{array}
$$

3.2 Translation of the Protocol State Machine

Now the protocol state machine for class Elevator has to be translated to CSP as well. While in general it is a non trivial task to translate a UML state machine to CSP, a simple translation scheme exists for state machines which obey the following constraints:

– simple events, no guards, no actions,
– no interlevel transitions,
– only completion transitions from compound states,
– disjoint event sets in concurrent submachines,
– no pseudo states besides initial states[6].

Many, if not most, *protocol* state machines already obey these constraints[7], so we do not regard them as severe restrictions in this context.

Translation. Let SM be the state machine. For any pseudo, simple or compound state s of SM let $\mathcal{C}_s$ denote the set of all direct successors of s with respect to completion transitions and $\mathcal{T}_s$ the set of all pairs (e,t) of direct successors t of s reached via a transitions triggered by e. For any submachine M of SM (including the top level state machine M_{top}) let $\mathcal{I}_M$ denote the set of initial states for M. Now a translation function φ from states and (sub-)machines to CSP process definitions can be defined[8]:

[5] The renaming keeps the CSP expression small and otherwise has no meaning.
[6] This implies that the history mechanism is not used.
[7] They often use guards, but in CSP-OZ enable schemas are used instead.
[8] The special processes $SKIP$ and $STOP$ represent termination and deadlock; $|||$ denotes parallel composition without synchronisation (interleaving) and $;$ denotes sequential composition.

$$\varphi(s) \equiv \begin{cases} P_s = SKIP & \text{if } s \text{ is a final state,} \\ P_s = \Box_{(e,t)\in\mathcal{T}_s} e \to P_t & \text{if } s \text{ is a simple state and } \mathcal{C}_s = \varnothing, \\ P_s = \sqcap_{t\in\mathcal{C}_s} P_t & \text{if } s \text{ is a simple state and } \mathcal{C}_s \neq \varnothing \\ & \text{or } s \text{ is an initial state,} \\ P_s = (|||_{i=1}^{n} P_{M_i});\ \big((\sqcap_{t\in\mathcal{C}_s} P_t) & \text{if } s \text{ is a compound state with} \\ \qquad \not< \mathcal{C}_s \neq \varnothing \not> STOP\big) & \text{submachines } M_i,\ 1 \leq i \leq n. \end{cases}$$

$$\varphi(M) \equiv \quad P_M = \sqcap_{t\in\mathcal{I}_M} P_t \qquad \text{for any (sub-) machine } M \text{ of } SM.$$

After calculating and combining the process expressions $\varphi(s)$ and $\varphi(M)$ for all states s and all (sub-) machines M of SM, the CSP semantics of SM can now be computed by evaluating $P_{M_{\text{top}}}$ using one of the CSP models.

Since the protocol state machine in Fig. 2) satisfies the constraints given above, it can be translated using the scheme above. This yields the following process definitions (to the right an equivalent simplified version of each process body is shown):

$$\begin{aligned}
P_{M_{\text{top}}} &= \sqcap_{i\in\{1\}} P_i && = P_1 \\
P_{M_1} &= \sqcap_{i\in\{3\}} P_i && = P_3 \\
P_{M_2} &= \sqcap_{i\in\{9\}} P_i && = P_9 \\
P_1 &= \sqcap_{i\in\{2\}} P_i && = P_2 \\
P_2 &= (|||_{i\in\{1,2\}} P_{M_i});\ \big((\sqcap_{t\in\varnothing} P_t) && = (P_{M_1} \ ||| \ P_{M_2});\ STOP \\
&\quad \not< \varnothing \neq \varnothing \not> STOP\big) && \\
P_3 &= \sqcap_{i\in\{4\}} P_i && = P_4 \\
P_4 &= \Box_{(e,t)\in\{(\text{start},5)\}} e \to P_t && = \text{start} \to P_5 \\
P_5 &= \Box_{(e,t)\in\{(\text{closeDoor},6)\}} e \to P_t && = \text{closeDoor} \to P_6 \\
P_6 &= \Box_{(e,t)\in\{(\text{up},7),(\text{down},7)\}} e \to P_t && = \text{up} \to P_7 \ \Box \ \text{down} \to P_7 \\
P_7 &= \Box_{(e,t)\in\{(\text{passed},7),(\text{stop},8)\}} e \to P_t && = \text{passed} \to P_7 \ \Box \ \text{stop} \to P_8 \\
P_8 &= \Box_{(e,t)\in\{(\text{openDoor},4)\}} e \to P_t && = \text{openDoor} \to P_4 \\
P_9 &= \sqcap_{i\in\{10\}} P_i && = P_{10} \\
P_{10} &= \Box_{(e,t)\in\{(\text{request?f},10)\}} e \to P_t && = \text{request?f} \to P_{10}
\end{aligned}$$

With $PROC_{SM} = P_{M_{\text{top}}}$ we now have a CSP translation of the state machine part of the specification. Simplifying it again, the readable (but equivalent) version of $PROC_{SM}$ looks like this:

$$\begin{aligned} \mathit{PROC}_{\mathit{SM}} &= P_4 \;|||\; P_{10} \\ P_4 &= \text{start} \rightarrow \text{closeDoor} \rightarrow (\text{up} \rightarrow P_7 \;\square\; \text{down} \rightarrow P_7) \\ P_7 &= \text{passed} \rightarrow P_7 \;\square\; \text{stop} \rightarrow \text{openDoor} \rightarrow P_4 \\ P_{10} &= \text{request?f} \rightarrow P_{10} \end{aligned}$$

3.3 Resulting Specification

As a last step in the translation of the Object-Z class and its protocol state machine the processes obtained for each one are put in parallel

$$\mathit{PROC} \quad = \quad \mathit{PROC}_Z \underset{\Sigma}{\|} \mathit{PROC}_{\mathit{SM}}$$

synchronising on the set Σ of all events (methods, including parameters) specified in the Object-Z class using `method` or `chan` declarations. In this case parallel composition can be viewed as a conjunction, that is PROC accepts a method call (sent or received) iff both PROC_Z and $\mathit{PROC}_{\mathit{SM}}$ accept it. This is the intended semantics of the combined specifications with respect to UML protocol state machines, i.e., the protocol state machine restricts possible behaviour.

4 Notions of Consistency

Using the example given above we now discuss different notions of consistency for specifications consisting of Object-Z classes and protocol state machines. We only refer to the result of the translation, the semantics of PROC.

What does consistency mean in our context? Informally, consistency here is about how the explicit specification of sequences of method invocations in the state machine and the implicit specification through the enabling conditions in the Object-Z part fit together. Formally, it is some property of PROC. In the sequel we present several possible definitions of consistency and for each of them develop a technique for proving it using the FDR model checker (in case of finite state specification)[9].

4.1 Basic Consistency

When talking about consistency it is beneficial to view parallel composition of CSP processes as conjunction. So the consistency of PROC is the consistency of '$\mathit{PROC}_Z \wedge \mathit{PROC}_{\mathit{SM}}$'. It is immediately clear that if this 'formula' is not satisfiable, the corresponding specification is *inconsistent*. Translated to the terms of the semantic model this means: PROC will always deadlock, i.e. any sequence of events offered to PROC will lead to deadlock. Seen the other way round: for the specification to be satisfiable it suffices to have at least one trace of PROC not leading to deadlock.

[9] We only consider reactive systems without termination; to use these for systems which include finite behaviour, termination has to be mapped to an infinite iteration of some extra event.

Definition 1. *A specification consisting of an Object-Z class and an associated state machine has the property of* satisfiability *iff the corresponding process in the semantic model has at least one non-terminating run.*

This property can be automatically checked as follows. We assume Σ to be the alphabet of the class (i.e. all events occurring in $PROC$), the event acc not to be in Σ, and $PROC$ to be given in machine readable CSP so that the FDR model checker can be used. In CSP, properties are checked by comparing the system process with another CSP process specifying the property. The comparison is a *refinement test* in one of the models of CSP: traces, failures or failures and divergences. For our first property we use the following property process INF, parameterised in the event (or method) m under consideration:

$$INF(m) = m \rightarrow INF$$

For checking satisfiability we test whether

$$PROC[acc/m] \sqsubseteq_{\mathcal{T}} INF(acc)$$

holds. The process $INF(acc)$, only executing acc events, is a trace refinement of $PROC$ in which all events are renamed to acc if $PROC$ contains at least one nonterminating run. Performing this check for our elevator example tells us that the specification is satisfiable.

Satisfiability does, however, not exclude the case that the specification deadlocks. Since in general deadlock occurs several steps after performing the 'wrong' event, successful usage of the system specified would amount to guessing one trace from the infinite set of traces. This is clearly not a sufficient form of consistency.

For a specification to have basic consistency we thus require $PROC$ to be deadlock free; after any sequence of events performed by $PROC$ there has to be at least one event to continue the sequence, that is, no trace may have Σ (the set of all methods) as the refusal set. This can be regarded as the standard notion of consistency in the context of behavioural specifications and is used by other authors as well, for instance [7, 6], where it is applied to the behaviour of *different* entities of a model acting together.

Definition 2. *A specification consisting of an Object-Z class and an associated state machine has the property of* basic consistency *iff the corresponding process in the semantic model is* deadlock free.

Using the FDR model checker for CSP it can now be checked whether the example is consistent according to Def. 2 (by using FDR's predefined test for deadlock-freedom). The result is, that $PROC$ for the example given is indeed deadlock free, so the specification has the property of basic consistency.

4.2 Execution of Methods

Now that we have defined a first form of consistency we again look at our example to see whether this is sufficient. Consider the following sequence of events on $PROC_Z$ starting from the initial state.

$$\begin{array}{rcll}
 & & Z(0,0,\varnothing) & \\
\mathit{request}.1 & \rightarrow & Z(0,0,\{1\}) & \\
\mathit{start} & \rightarrow & Z(0,1,\varnothing) & [\,\mathit{requests} \neq \varnothing\,] \\
\mathit{closeDoor} & \rightarrow & Z(0,1,\varnothing) & [\,\mathit{pos} \neq \mathit{tar}\,] \\
\mathit{up} & \rightarrow & Z(0,1,\varnothing) & [\,\mathit{pos} < \mathit{tar}\,] \\
\mathit{request}.1 & \rightarrow & Z(0,1,\{1\}) & \\
\mathit{passed} & \rightarrow & Z(1,1,\{1\}) & [\,\mathit{pos} \neq \mathit{tar}\,] \\
\mathit{stop} & \rightarrow & Z(1,1,\{1\}) & [\,\mathit{pos} = \mathit{tar}\,] \\
\mathit{openDoor} & \rightarrow & Z(1,1,\{1\}) & [\,\mathit{pos} = \mathit{tar}\,] \\
\mathit{start} & \rightarrow & Z(1,1,\varnothing) & [\,\mathit{requests} \neq \varnothing\,]
\end{array}$$

At this point the elevator should be able to perform *closeDoor*, but this method is only enabled if $\mathit{pos} \neq \mathit{tar}$, so the only method enabled at this point is *request*. Since *request* does not modify *pos* or *tar* this condition will endure infinitely. This means, the only possible trace after this prefix is $\langle \mathit{request} \rangle^*$. The specification has the property of basic consistency but still the combination of state-based part and protocol state machine prevents certain executions which we expect from the elevator. What we additionally need are certain forms of *liveness* of methods.

As a first approach we might require that every method specified in the interface of the class can be executed at least once.

Definition 3. *A specification consisting of an Object-Z class and an associated state machine has the property of* method executability *iff in the corresponding process in the semantic model every method is executed at least once.*

Executed at least once means that there is some trace in which an event corresponding to the method occurs. In FDR this can be checked as follows. We define a tester process

$$\mathit{ONCE}(m) = m \rightarrow \mathit{STOP}$$

and check the trace inclusion

$$\mathit{PROC} \setminus (\Sigma \setminus \{m\}) \sqsubseteq_{\mathcal{T}} \mathit{ONCE}(m)$$

This test has to be carried out for every method in the interface of the class. Since in the above example trace of the elevator method *closeDoors* has already been executed, this definition does however not have the desired effect: Our specification has the property of method executability for all methods in the interface of *Elevator*.

Although method executability is a weak requirement on specifications, it is a fundamental one: it amounts to the detection of 'dead code' in the specification, i.e., if a method m fails this test, anything can be substituted for it without changing the semantics of the specification. This is almost always an error.

A stronger definition might require that all methods (or at least certain methods marked to be live) should be executed infinitely often:

Definition 4. *A specification consisting of an Object-Z class and an associated state machine has the property of* method liveness *iff in the corresponding process in the semantic model every method will always eventually be executed again.*

This requirement is close to the definition of *impartiality* of [17] (or unconditional fairness of [13]) which states that every process in a concurrent system takes infinitely many moves. Clearly, method liveness is not fulfilled by our example. It can be tested using again the property process *INF* defined above:

$$\mathit{INF}(m) \sqsubseteq_{\mathcal{FD}} \mathit{PROC} \setminus (\Sigma \setminus \{m\})$$

There is still some drawback in this strong liveness requirement. For object-oriented systems it is not adequate to require that methods are always executed since an execution requires a request from the environment, and there might well be traces on which a method is never executed simply because it is never requested. What we really would like to have is liveness with respect to an *offering* of methods to a client of the class.

Definition 5. *A specification consisting of an Object-Z class and an associated state machine has the property of* method availability *iff in the corresponding process in the semantic model every method will always eventually be enabled again.*

Unfortunately, this kind of unconditional liveness is not expressible within the failures-divergences model. We have to approximate it by a form of *bounded* availability, fixing an upper bound N on the number of steps in between two offerings of a method.

$$\begin{aligned} \mathit{OFFER}(i,m) = {} & \big(\textstyle\bigsqcap_{Ev \subseteq \Sigma} \Box_{ev \in Ev \cup \{m\}}\, ev \to \mathit{OFFER}(N,m)\big) \\ & \Box\ i > O\ \&\ \textstyle\bigsqcap_{Ev \subseteq \Sigma \setminus \{m\}} \Box_{ev \in Ev}\, ev \to \mathit{OFFER}(i-1,m) \end{aligned}$$

After at most N steps process $\mathit{OFFER}(N,m)$ reaches a state in which method m is not refused. Regarding the other events the process can freely choose to refuse as well as execute them. The check for bounded method availability of m is then

$$\mathit{OFFER}(N,m) \sqsubseteq_{\mathcal{FD}} \mathit{PROC}$$

A last remark on this test concerns efficiency. Since process *OFFER* contains a choice over all possible subsets of Σ the state space of *OFFER* will be exponential in the size of the class' alphabet. For larger specifications this might make it unrealistic to actually carry out the check. Fortunately, for this test all events besides m in $\mathit{OFFER}(N,m)$ and *PROC* can be regarded as equivalent, so renaming can be used to reduce the size of the alphabet to two, without changing the outcome of the test. Giving the reduced form as actual input to FDR results in clearly reduced runtime comparable to the other checks.

Summarising, we have proposed and discussed five definitions of consistency which can be used for classes and associated state machines. Which ones are actually used depend on the designer of the specification. In our opinion, basic consistency and method executability should always should always be fulfilled. The extended liveness conditions are in general not to be used for all methods, since some kinds of methods, e.g., those performing initialisation, are not intended to have these properties in the first place. Since this cannot be inferred from the specification as it is now, it might be useful to include additional elements in the diagrams to indicate the intentions of the modeller with respect to the intended behaviour.

5 Consistency-Preserving Transformation

Having established consistency in early phases of system development it is desirable to preserve it during successive model evolutions. In a formal approach to system development model evolutions from high-level specifications to lower-level ones are supported by the concept of refinement [4]. Refinement defines correctness criteria for allowed changes between different levels of abstraction. With regard to consistency refinement should preserve consistency, or rather the other way round: consistency should be defined such that it is preserved under refinement. In the sequel we will examine which of our five definitions of consistency are preserved under refinement.

Since we have a state-based and a behaviour-oriented part there are in principle two kinds of refinement to be considered: *process refinement* in CSP, defined as inclusion in the failures-divergences model, and *data refinement*, proven via simulation rules between classes. Fortunately, data refinement in Object-Z induces failures-divergences refinement on its CSP semantics [20, 15, 10]. Hence it is sufficient to study preservation of consistency under process refinement. In the sequel we let refinement stand for failures-divergences refinement.

We pass through our definitions in the order in which they are defined in the last section. For the first one we get:

Proposition 1. *Satisfiability is not preserved under refinement.*

This fact can be illustrated by the following CSP process P over the alphabet $\Sigma = \{a, b, c\}$ representing some process $PROC$:

$$\begin{aligned} P &= a \rightarrow STOP \\ &\quad \sqcap\ b \rightarrow Run \\ Run &= \Box_{ev \in \Sigma \setminus \{a\}}\ ev \rightarrow Run \end{aligned}$$

P has the property of satisfiability: a nonterminating run is $b, c, c, c, \ldots$. Consider now the process P' defined as $a \rightarrow STOP$. P' is a failures-divergences refinement of P (due to the internal choice at the start of P), but is has no nonterminating run.

Concerning basic consistency we get a better result. The following proposition follows from standard CSP theory.

Proposition 2. *Basic consistency is preserved under refinement.*

Concerning the execution of methods there is one negative and two positive results.

Proposition 3. *Method executability is not preserved under refinement.*

Starting with the same process P which we used for the counterexample about satisfiability we now refine it to a process $P'' = b \rightarrow Run$. Process P has the property of method executability for all methods in Σ, including a. P'' is a refinement of P but allows for no execution of a.

The stronger requirements of method liveness and availability are however preserved:

Proposition 4. *Method liveness and method availability are preserved under refinement.*

Proof. Referring to the processes used for checking these two requirements the results easily follow from transitivity of refinement and monotony of all CSP operators with respect to refinement.

Together with the discussion in the last section this gives strong hints as to what a reasonable definition of consistency might be. Basic consistency should always be fulfilled in order to achieve a meaningful model. This should be complemented with liveness and/or availability of methods, where, however, it might make sense to restrict these requirements to some of the methods.

For the state machine part of a model we can then classify one kind of consistency preserving transformations via refinements: whenever we replace a state machine SM1 by a state machine SM2, consistency is preserved if the CSP process belonging to SM2 is a process refinement of that of SM1. An interesting point for further research would be to find classes of transformations on state machines which induce refinements, as for instance [5] shows the connection between some notions of statechart inheritance and refinement. For the state-based part of the model (the class) data refinement is a consistency preserving transformation.

6 Conclusion

In this paper we discussed consistency for specifications consisting of an Object-Z class describing the data aspects of a class and an associated state machine describing the allowed sequences of method calls. By means of a translation to a common semantic domain a semantics was given for the whole specification, which enabled a number of consistency definitions. For every such definition we proposed a technique for automatically checking it with a model checker, and we furthermore showed which of the definitions are preserved under refinement.

Related Work. For the UML, consistency is a heavily discussed topic, see for instance the workshop on "Consistency Problems in UML-based Software Development" [16]. The approaches in this workshop discuss a wide variety of consistency issues arising in UML, ranging from mainly syntactic ones to others involving a semantic analysis of the model. The work closest to ours is that of [7, 6] who also use CSP as their semantic basis (and FDR for model checking). However, whereas they are comparing different behavioural views of a UML model (state machines and protocols in UML-RT) we define consistency between a static and a dynamic diagram.

The basis for our work is CSP-OZ, an integration of Object-Z and CSP, which – together with the translation of state machines to CSP – allows for a common analysis of state-based and behaviour-oriented views. Besides CSP-OZ there are a number of other integrated specification languages around, for an overview and comparison of integrations of Z and process algebras see [9].

Consistency of different views is (or has been) an issue in other areas as well, especially in the ODP ISO reference model which allows for a specification of distributed systems by different viewpoints. The approaches taken there are, however, different from ours. [8, 1] achieve consistency by transformations between viewpoints, [2] define consistency between viewpoints by the existence of a common implementation of both viewpoints (using a variety of possible implementation/refinement relations). The latter approach has also been taken by Davies and Crichton for defining consistency between sequence diagrams and system models in UML [3] (also using CSP as a semantic domain). While for the comparison of sequence diagrams specifying certain scenarios of a system a refinement based consistency definition seems reasonable (in order to find out whether the system sometimes/always exhibits the scenario), for class and state diagrams it might turn out to be inadequate: Since both the traces model and the stable failures model of CSP have top elements with regard to the respective refinement order, a common refinement always exists, whereas a specification, which is consistent according to some of our definitions, typically does not have a common refinement in the failures-divergences model. Moreover, in our opinion our consistency definitions more naturally capture a practitioners point of view on consistency.

References

1. C. Bernardeschi, J. Dustzadeh, A. Fantechi, E. Najm, A. Nimour, and F. Olsen. Transformations and Consistent Semantics for ODP Viewpoints. In *Proceedings of Second IFIP conference on Formal Methods for Open Object-based Distributed Systems - FMOODS'97*. Chapman & Hall, 1997.
2. H. Bowman, M.W.A. Steen, E.A. Boiten, and J. Derrick. A formal framework for viewpoint consistency. *Formal Methods in System Design*, 21:111–166, 2002.
3. J. Davies and Ch. Crichton. Concurrency and Refinement in the Unified Modeling Language. *Electronic Notes in Theoretical Computer Science*, 70(3), 2002.
4. J. Derrick and E. Boiten. *Refinement in Z and Object-Z, Foundations and Advanced Application*. Springer, 2001.
5. G. Engels, R. Heckel, and J. Küster. Rule-based Specification of Behavioral Consistency based on the UML Meta-Model. In Martin Gogolla, editor, *UML 2001*. Springer, 2001.

6. G. Engels, R. Heckel, J. M. Küster, and L. Groenewegen. Consistency-preserving model evolution through transformations. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002 – Model Engineering, Concepts, and Tools*, volume 2460 of *LNCS*, pages 212–226. Springer-Verlag, 2002.
7. G. Engels, J. Küster, R. Heckel, and L. Groenewegen. A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models. In *9th ACM Sigsoft Symposium on Foundations of Software Engineering*, volume 26 of *ACM Software Engineering Notes*, 2001.
8. K. Farooqui and L. Logrippo. Viewpoint Transformation. In *Proc. of the International Conference on Open Distributed Processing*, pages 352–562, 1993.
9. 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.
10. C. Fischer and S. Hallerstede. Data-Refinement in CSP-OZ. Technical Report TRCF-97-3, University of Oldenburg, June 1997.
11. C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of the 1st International Conference on Integrated Formal Methods (IFM)*, pages 315–334. Springer, 1999.
12. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, Oct 1997.
13. N. Francez. *Fairness.* Texts and Monographs in Computer Science. Springer, 1986.
14. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
15. M.B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.
16. L. Kuzniarz, G. Reggio, J. L. Sourrouille, and Z. Huzar, editors. *UML 2002 – Workshop on Consistency Problems in UML-based Software Development*, volume 06 of *Blekinge IOT Research Report*, 2002.
17. D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, Justice and Fairness: The Ethics of Concurrent Termination. In G. Goos and J. Hartmanis, editors, *Automata, Languages and Programming*, number 115 in LNCS, pages 264 –277. Springer, 1981.
18. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1997.
19. G. Smith. *The Object-Z Specification Language.* Kluwer Academic Publishers, 2000.
20. G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In M. Hinchey and Shaoying Liu, editors, *Int. Conf. of Formal Engineering Methods (ICFEM)*, pages 293–302. IEEE, 1997.
21. OMG Unified Modeling Language specification, version 1.5, March 2003. http://www.omg.org.

Compositional Verification Using CADP of the ScalAgent Deployment Protocol for Software Components*

Frédéric Tronel, Frédéric Lang, and Hubert Garavel

INRIA Rhône-Alpes / VASY
655, avenue de l'Europe
F-38330 Montbonnot, France
{Frederic.Tronel,Hubert.Garavel,Frederic.Lang}@inria.fr

Abstract. In this article, we report about the application of the CADP verification toolbox to check the correctness of an industrial protocol for deploying and configuring transparently a large set of heterogeneous software components over a set of distributed computers/devices. To cope with the intrinsic complexity of this protocol, compositional verification techniques have been used, including incremental minimization and projections over automatically generated interfaces as advocated by Graf & Steffen and Krimm & Mounier. Starting from the XML description of a configuration of components to be deployed by the protocol, a translator produces a set of LOTOS descriptions, μ-calculus formulas, and the corresponding compositional verification scenario to be executed. The approach is fully automated, as formal methods and tool invocations are made invisible to the end-user, who only has to check the verification results for the configuration under study. Due to the use of compositional verification, the approach can scale to large configurations. So far, LOTOS descriptions of more than seventy concurrent processes have been verified successfully.

1 Introduction

Formal verification methods are a key approach to establish the correctness of complex and critical object-oriented systems. This is true for sequential systems, and even more true for concurrent systems in which objects execute and interact using several threads of control.

However, the complexity of a system grows fast as the number of objects increases, so that attempts at verifying real-life systems are quickly confronted to the *state explosion* problem. It is therefore of crucial interest to focus on verification methods that scale up appropriately when applied to systems of increasing complexity.

Compositional verification methods usually follow a *divide and conquer* approach. The system to be verified is decomposed in several components, which

* This work was partially funded by the French Ministry of Industry under RNTL contract "PARFUMS".

are analyzed separately; afterwards, the results of the separate analyses are combined together to analyze the whole system. There are different approaches to compositional verification, depending whether the system under analysis is sequential or concurrent, and in the latter case, depending on the semantics used to model concurrency: linear-time or branching-time, state-based or action-based, etc. There is an important corpus of literature on compositional verification; for a survey, see for instance [19], [11, Section 1.1], [4, Sections 1.1 and 1.2], etc.

In spite of the many publications on compositional verification, the number of real-life case-studies in which compositional techniques have been applied is still low. This number is even lower if one considers the case of object-based systems, since compositional verification has been so far mostly used for communication protocols [20, 10, 14] or hardware protocols [3]. As for object-based systems specifically, one can mention several lines of work. [2] uses compositional proofs and refinement techniques to verify one-to-many negotiation processes for load balancing of electricity use. [1] uses a compositional proof system to verify correctness properties (expressed using the modal μ-calculus) for a set of applets executing on open platforms.

The present article is different, as it relies on enumerative (a.k.a., explicit state) model checking rather than proof techniques. It builds upon a prior application of compositional verification [6] to a dynamic reconfiguration protocol for a middleware agent-based platform. Using the CADP [9] verification toolbox, it was possible to establish the correctness of the reconfiguration protocol for several finite configurations determined by the number of agents, execution sites and protocol primitives. However, the approach did not scale well to larger configurations, mainly because the architecture of the system was specified in a centralized manner, all agents being connected to a central process modeling (an abstraction of) the software bus provided by the middleware infrastructure. This central process — more or less similar to a FIFO queue — prevented compositional verification from scaling up, as it was not possible to generate its entire state space in isolation from the remainder of the system. One key conclusion of [6] was the need for a more decentralized architecture specification in order to improve scalability.

Precisely, this research direction is addressed in the present article, still in the framework of industrial middleware infrastructures although on a different case-study than [6]. We consider here a deployment protocol for software components, which is commercialized by the SCALAGENT software company[1], and which we analyze using the compositional verification tools of CADP.

The present article is organized as follows. Section 2 recalls the principles of compositional verification techniques and explains how they are supported within the CADP toolbox. Section 3 describes the essential features of the SCALAGENT deployment protocol. Section 4 gives hints of the formal modeling of the deployment protocol and indicates how the modeling task was, to a large extent, automated. Section 5 presents the main results of compositional verification. Finally, Section 6 concludes the article.

[1] `http://www.scalagent.com`

2 Compositional Verification with CADP

In this section, we first present enumerative and compositional verification techniques for systems composed of asynchronous processes, and we then detail how these techniques are supported by the CADP verification toolbox.

Given a formal specification (e.g., using the ISO formal description technique LOTOS [13]) of a concurrent system to be verified, enumerative verification relies on the systematic exploration of (some or all) possible executions of the system. The set of all possible executions can be represented as an LTS (*Labeled Transition System*), i.e., a graph (or state space) containing *states* and *transitions* labeled with communication actions performed by concurrent processes. There are two approaches to enumerative verification:

– In the first way, an *explicit* LTS is generated, i.e., states and transitions are first enumerated and stored, then analyzed by verification algorithms.
– In the second way, an *implicit* LTS (consisting of an initial state and a function that computes the successors of a given state) is constructed and verified at the same time, the construction being done *on the fly* depending on the verification needs. This allows to detect errors before the LTS has been generated entirely.

For complex systems, both approaches are often limited by the *state explosion* problem, which occurs when state spaces are too large for being enumerated. Two *abstraction* mechanisms are of great help when attacking state explosion:

– *Communication hiding* permits to ignore communication actions that need not be observed for verification purpose;
– *Minimization* (with respect to various equivalence relations, such as strong bisimulation, branching bisimulation, etc.) allows to merge LTS states with identical futures and (possibly) to collapse sequences of hidden communication actions.

A further step is compositional verification, which consists in generating the LTS of each concurrent process separately, then minimizing and recombining the minimized LTSs taking advantage of the congruence properties of parallel composition. The joint use of hiding and minimization to reduce intermediate state spaces enables to tackle large state spaces that could not be generated directly.

Although this simple form of compositional verification has been applied successfully to several complex systems (e.g., [3]), it may be counter-productive in other cases: generating the LTS of each process separately might lead to state explosion, whereas the generation of the whole system of concurrent processes can succeed if processes constrain each other when composed in parallel.

This issue has been addressed in various refined compositional verification approaches, which allow to generate the LTS of each separate process by taking into account *interface constraints* representing the behavioral restrictions imposed on each process by synchronization with its neighbor processes. Taking into account the environment of each process allows to eliminate states and transitions that are not reachable in the LTS of the whole system.

CADP[2] (*Construction and Analysis of Distributed Processes*) is a widely spread toolbox for protocol engineering, which offers a large range of functionalities, including interactive simulation, formal verification, and testing. CADP was originally dedicated to LOTOS, but its modular architecture makes it open to other languages and formalisms. CADP contains numerous tools for the compositional verification of complex specifications written in LOTOS:

– As regards explicit LTS representation, CADP provides a compact graph format, BCG (*Binary Coded Graph*) together with code libraries and tools to create, explore, and visualize BCG graphs, and to translate them from/to many other graph formats.
– As regards implicit LTS representation, CADP provides an extensible environment named OPEN/CÆSAR [7]. Although independent from any particular specification language, OPEN/CÆSAR has compilers for several languages: LOTOS (CÆSAR), explicit LTSs (BCG_OPEN), networks of communicating LTSs (EXP.OPEN), etc.). The code generated by OPEN/CÆSAR compilers is used by on the fly algorithms to perform simulation, verification, test generation, etc. on implicit LTSs.
– As regards LTS generation from LOTOS descriptions, CADP provides the CÆSAR and CÆSAR.ADT compilers, which can compile a LOTOS specification (or particular processes in this specification).
– As regards parallel composition of LTSs, CADP provides the EXP.OPEN compiler for handling networks of communicating LTSs, connected using LOTOS parallel composition and communication hiding operators.
– As regards communication hiding, CADP provides the BCG_LABELS tool, which allows to hide and/or rename the communication actions of an LTS.
– As regards LTS minimization, CADP contains two tools: BCG_MIN, which performs strong and branching minimization of LTSs efficiently, and ALDÉBARAN, which implements additional equivalences (safety equivalence, observational equivalence, and tau*.a equivalence) and LTS comparison algorithms.
– As regards generation with interface constraints, CADP provides the PROJECTOR tool, which implements the refined compositional verification approach of [12, 15]. PROJECTOR can be used to restrict LOTOS processes, explicit LTSs, as well as networks of communicating LTSs.
– As regards modeling of asynchronous communication media, we added a new tool named BCG_GRAPH, which generates various classes of useful LTSs, such as FIFO buffers and bags[3]. Distributed systems often contain many occurrences of such buffers that differ only by a few parameters, such as size and message names. Using BCG_GRAPH, very large buffers (several hundreds thousands states) can be generated quickly, with a small memory footprint.

[2] `http://www.inrialpes.fr/vasy/cadp`

[3] A bag is a fully asynchronous buffer that does not preserve message ordering.

– Finally, CADP includes a scripting language named SVL [8, 16], which provides a high-level interface to all the aforementioned CADP tools, thus enabling an easy description of complex compositional verification scenarios. For instance, the scenario consisting in generating separately the LTSs of three processes P, Q, and R contained in a file named "spec.lotos", minimizing them for branching bisimulation, composing them in parallel, minimizing the resulting network on the fly, and storing the resulting LTS in the BCG format in a file named "PQR.bcg", can be described by the simple SVL script that follows:

  ```
  % DEFAULT_LOTOS_FILE="spec.lotos"
  "PQR.bcg" = root leaf branching reduction of P || Q || R;
  ```

  The SVL compiler translated such an SVL script into a Bourne shell script that, when executed, invokes CADP tools in the appropriate order and stores the results in intermediate files.
  SVL has many additional features, namely operations to restrict a system with respect to a given interface, to minimize systems with respect to several other bisimulations and equivalences, to hide and rename communication actions, and statements to verify temporal logic formulas, to check for deadlocks and livelocks, and to compare systems with respect to a given equivalence or bisimulation. SVL scripts can also contain Bourne shell constructs, which enables to mix, e.g., conditionals, loops, and calls to Unix commands with SVL statements.

3 The ScalAgent Deployment Protocol

The work presented in this article was funded in the scope of PARFUMS (*Pervasive Agents for Reliable and Flexible Ups Management Services*), an industrial research project involving three companies (MGE-UPS, SILICOMP Research Institute, and SCALAGENT Distributed Technologies), and the VASY research group at INRIA.

The goal of PARFUMS is to solve problems of UPS (*Uninterruptible Power Supply*) management (installation, repair, and monitoring of remote equipments) in the case of large scale sites, by embedding software within UPSs. To master the complexity induced by the distribution of applications, the project relies on the SCALAGENT platform for embedded systems, written in JAVA, to configure, deploy, and reconfigure software. Our contribution is about the modeling of the SCALAGENT deployment protocol and its verification using the CADP toolbox.

To ensure scalability, the SCALAGENT deployment protocol relies on a tree-like hierarchy of distributed agents communicating asynchronously by the mean of events. The tree of agents is meant to reflect the geographical distribution of software components to be deployed. The protocol uses two kinds of agents:

– *Containers* are located at the leaves of the tree hierarchy. They encapsulate software components written in any language, and act as interfaces with the rest of the protocol.

- *Controllers* are located at higher nodes of the tree hierarchy and manage the deployment. They only communicate with their parent and children agents, which allows a significant reduction of communications.

Controllers are specified by a workflow of *activities*, themselves structured as a tree. Activities fall into three categories, depending on the way they spawn subactivities:

- *Elementary activities* are simple tasks that do not involve other subactivities. An example of such an activity is the receipt of a particular event from a container to signal its successful deployment, followed by an appropriate reaction.
- *Sequential activities* can spawn a set of subactivities, each one being executed after the previous one terminates.
- *Parallel activities* can spawn a set of simultaneous subactivities.

Beside activities, there exists a central *referential* process gathering information sent by the elementary activities regarding the success or failure of the deployment. Given this data, the referential process decides whether deployment is possible or not. The existence of communications between elementary activities and the referential "slightly breaks" the tree structure of communications. This is illustrated on Figure 1, the referential process being the white node in the "tree" of activities.

To describe distributed configurations, the ScalAgent infrastructure relies on the use of an Xml Dtd named Xolan. An Xolan configuration describes a set of controllers and containers, their geographical distribution, as well as their dependencies in terms of provided and required services, which rule the way the deployment must be performed. Xolan is generic, that is, not specific to Ups management. A graphical interface allows to specify a hierarchy of Upss and software to be deployed and generates the corresponding Xolan configuration automatically (see Figure 1).

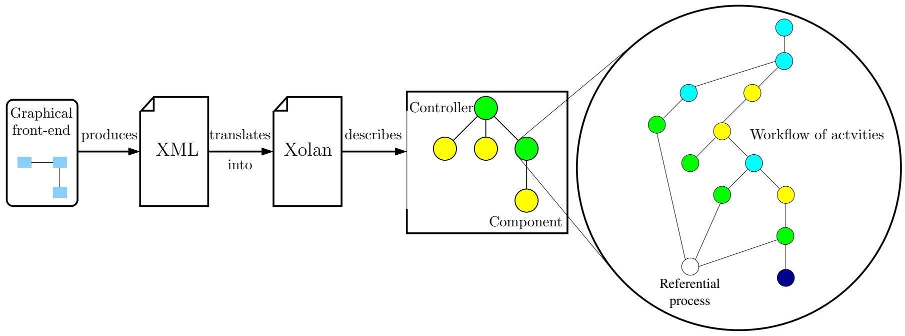


Fig. 1. Different description levels for the deployment protocol.

4 Automated Formal Modeling of Configurations

It would have been possible to model XOLAN configurations using LOTOS abstract data types and to specify the deployment protocol as a LOTOS process parameterized by the configuration to be deployed. However, this would have required the dynamic creation of processes in function of the configuration, which is not supported by mainstream enumerative verification tools.

Instead, we chose an automated approach by developing a translator (11, 000 lines of the object-oriented, functional language OCAML [17]) that takes an XOLAN configuration, and produces both the LOTOS specification corresponding to this configuration and an SVL script to perform the verification. This approach meets several requirements:

– Dynamic creation of processes is avoided, since the OCAML translator can statically determine the set of processes created by the protocol for a given configuration.
– XML parsing and XOLAN data structure handling are delegated to the OCAML translator rather than being coded as LOTOS abstract data types.
– Even for simple configurations, the corresponding LOTOS specifications and SVL scripts are complex, due to the large number of concurrent processes. The existence of an automated translator allows to propagate changes in the protocol modeling to each configuration under study so as to maintain consistency.

To keep the formal verification of the protocol as intuitive as possible, each activity in the specification is translated into a separate process in the generated LOTOS code. This way, an incorrect behaviour in a given process can be immediately tracked back to the corresponding activity.

The processes generated for all activities share a similar form shown on Figure 2. Each process communicates with other parts of the system using three gates named `SEND`, `RECV`, and `ERROR`:

– "`SEND !from !to !event`" is used by the process whose identifier is stored in variable "`from`". It indicates that message "`event`" should be sent to the process whose identifier is stored in variable "`to`".
– "`RECV ?from:PID !to ?event:EVENT`" is used by the process whose identifier is stored in variable "`to`". Once the receipt is done, the variable "`from`" of type `PID` will contain the identifier of the sending process and the variable "`event`" of type `EVENT` will contain the received event.
– "`ERROR !from !number`" is used by the process whose identifier is stored in variable "`from`" to indicate that the error referenced as "`number`" has occurred.

Each process is recursive and stores its current state in a parameter "`state`" of type `STATE`. Each computation step of a process consists of message receipt, followed by some reaction depending on the identity of the message sender, the event received, and the current process state (the LOTOS construct "`[...] ->`

...” reads as “*if ... then ...*”). A reaction consists in sending zero or more messages, followed by a state change, which is expressed by a recursive process call with actualized state. Some combinations of sender, event, and state may trigger an error message.

```
process P1 [SEND, RECV, ERROR] (state:STATE) : noexit :=
   RECV ?from:PID !P1 ?event:EVENT ;
   [from eq P3] -> (
      [event eq E_START] -> (
         [state eq INIT] -> (
            SEND !P1 !P4 !START ;
            SEND !P1 !P6 !START ;
            BehaviourP1 [SEND, RECV, ERROR] (RUN)
         )
         []
         [not (state eq INIT)] -> ERROR !P1 !E1
      )
      []
      [event eq E_STOP] -> ( ... )
      []
      [not ((event eq E_START) or (event eq E_STOP))] -> ERROR !P1 !E2
   )
   []
   [from eq P5] -> ( ... )
   []
   [not ((from eq P3) or (from eq P5))] -> ERROR !P1 !E3
endproc
```

Fig. 2. A LOTOS process following the asynchronous communicating process model.

The SCALAGENT protocol specification is strongly object-oriented as it was written to prepare the way for a JAVA implementation of the protocol. All activities belong to an abstract “activity” class, which is refined into three abstract subclasses corresponding to elementary, sequential, and parallel activities respectively. Each of these abstract subclasses has itself concrete subclasses (for instance, deployment activities are a subclass of parallel activities).

The behaviour of an activity is a transition function obtained by combining the attributes specific to this activity (such as the number of subactivities for sequential and parallel activities, a unique identifier of the activity, a list of possible events, etc.) with the methods belonging to the activity class or transitively inherited from superclasses. Inheritance has the effect of adding reactions to new combinations of process parameters (events, states, process identifiers, etc.). For a given configuration of the deployment protocol, inheritance can be solved at compile-time. Thus, the Lotos process corresponding to an activity can be synthesized statically from the methods defined in the activity class and superclasses. The use of an object-oriented language such as Ocaml avoids the need for writing an inheritance resolution algorithm explicitly: by implementing the activity class hierarchy with a similar Ocaml class hierarchy, the resolution of inheritance is automatically performed by the Ocaml compiler.

5 Compositional Verification of the Protocol

Compositional verification consists in decomposing the system under verification into smaller components that can be analyzed in isolation. There are often several ways of modeling and decomposing a given protocol; the feasibility and efficiency of compositional verification crucially relies on a thorough study of components and their interactions. In this section, we explain our choices and their impact on verification.

5.1 Centralized vs. Distributed Communication Media

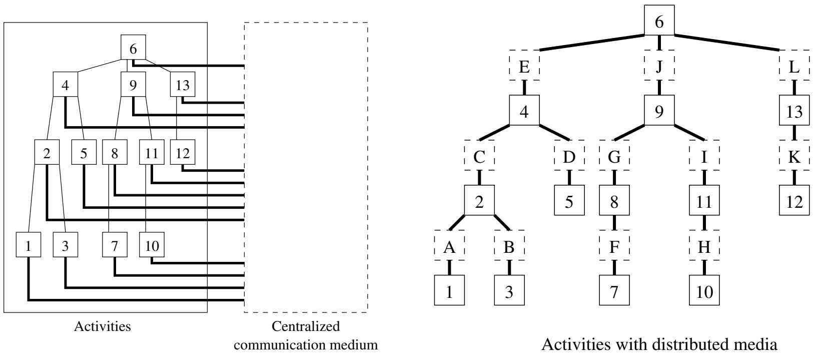
By design, the ScalAgent deployment protocol ensures that communications are pairwise between an activity and each of its subactivities, and that messages do not accumulate indefinitely in communication media (i.e., Fifo buffers and bags). Therefore communication media can be described as finite processes containing a bounded number of messages belonging to a finite set of possible values.

The protocol specification leaves a degree of freedom in the implementation of communication media: it does not specify whether communications are conveyed using one unique centralized communication medium or several distributed communication media.

A system with a centralized medium is schematically depicted on Figure 3(a). This was the approach followed in [6] to model a software bus between agents. Unfortunately, this approach cannot be reused for the deployment protocol. As the number of activities increases, the number of different messages that can be exchanged increases as well. As more activities introduce more asynchrony in the system, the number of messages that must be kept inside the medium also increases. Consequently, the state space of the centralized medium holding these messages may become too large for being generated separately.

In a refined approach, we split the centralized medium into one medium for each pair of communicating activities, as schematized on Figure 3(b). Each medium has to manage only a limited amount of communications, and is therefore less complex than the centralized medium.

This approach fits well with compositional verification, because the synchronizations between communicating processes are taken into account earlier in the verification process, leading to more constrained state spaces. Additionally, this permits to hide communications earlier, which, combined with minimization, gives greater opportunities to obtain reductions.



6
4
9
13
2
5
8
11
12
1
3
7
10
Activities
Centralized communication medium
E
J
L
C
D
G
I
K
A
B
F
H
Activities with distributed media

(a) Activities communicating through a centralized medium.

(b) Activities communicating through distributed media.

Fig. 3. Centralized vs. distributed media. Thick lines represent communications between activities and media, and thin lines represent the tree structure of activities.

As an example, the architecture of Figure 3(b), can be generated incrementally by generating the state spaces of medium A, activity 1, and activity 2 first, then composing them together with appropriate synchronizations. Then, the local communications between 1 and 2 (exchanged via medium A) can be hidden, and the resulting Lts minimized for branching bisimulation. The resulting system is then composed with activity 3 and medium B, hiding appropriate actions and minimizing the resulting Lts. This way, by incorporating media and processes in the numbering order, the system can be generated up to the root of the activity tree. The progressive application of hiding and minimization steps allows to keep a state space of tractable size in spite of the complexity introduced by parallel composition.

5.2 Communication Media Generation

In general, bounding the size of a communication medium may cause unexpected deadlocks or lost behaviours because of buffer overflows.

We addressed the issue by first generating (using Bcg_Graph) a medium of limited size (say, $N = 3$ places), which is composed in parallel with its connected activities. This parallel composition is then used as an interface to restrict (using Projector) the behaviour of the medium itself. This produces a subset of the state space corresponding to the medium, in which only the transitions synchronized with the activities are kept.

We then check whether the N places of the medium have been used in the composition with its related activities. This is done by checking (using the Evaluator model checker) whether there exists a sequence of N successive messages received by the medium. If not, no buffer overflow has occurred, which means that the buffer size was bounded correctly. Otherwise, an overflow might have occurred, and the experiment must be restarted after incrementing the value of N.

Figure 4 shows a fragment of SVL script implementing this technique. SVL statements are intertwined with Bourne shell statements (starting with the % character).

Although communication media could be expressed in LOTOS and translated to LTSs (as for the activities), the use of the dedicated BCG_GRAPH tool shortens the medium generation time by a factor of 10 to 100. This has a great impact on the overall verification time, since both the distributed media approach and the above technique to determine buffer size incrementally, require a significant number of media to be generated.

5.3 Additional Compositional Verification Tactics

More tactics have been used to make verification tractable:

- The referential process mentioned in Section 3 has been used as an interface to restrict (using the PROJECTOR tool) the behaviour of elementary activities communicating with this process. This divides by 2 the state space of some elementary activities. The referential process has been also used to restrict compositions of activities in several places.
- The state space of a process isolated from its context may explode if data communications are broken off without caution. For instance, the state space of the parallel composition of actions "$G\ ?X : \textbf{nat} \mid\mid G\ !1$" has a single transition labeled "$G\ !1$". However, the state space of "$G\ ?X : \textbf{nat}$" taken isolately cannot be generated because infinitely many different natural numbers can be received on gate G. For the deployment protocol, it is possible (though not easy) to determine statically the values exchanged by a process on gates `SEND` and `RECV`. We thus have improved the generated LOTOS code by adding communication guards (synthesized during a first phase of the translation) to constrain the set of potentially received data.

5.4 Results of Compositional Verification

The study of the protocol allowed to clarify (in accordance with the SCALAGENT designers) several obscure points in the protocol specification. In particular, the original specification was silent about the model of communications between activities. Model checking verification revealed (by exhibiting an infinite loop of error messages) that the protocol was not meant for handling asynchronous messages between the inner activities of a controller, and would function properly only if communications inside a controller are implemented as local procedure calls (i.e., the calling activity gets suspended until the procedure call returns). Consequently, communications within a controller can be modeled as FIFO buffers instead of bags. However, bags are still needed to model communications between controllers, which can be geographically distributed.

Figure 5 summarizes the verification results for four configurations. All experiments were done on a LINUX workstation with 1Gb memory and 2.2 GHz PENTIUM IV processor. We draw two main conclusions from these experiments:

```
% N=3
(* we know empirically that many media have at least 2 places
 * we start from N=3 (instead of N=1) to save time *)

% while true
% do

(* generation of a bag with $N places between processes 18 and 19 *)
% bcg_graph -bag $N LABELS_19_18.txt MEDIUM_19_18.bcg

"TMP.bcg" =
   branching reduction of
      gate hide all but RECV_19_18, SEND_19_18, RECV_18_19, SEND_18_19 in
         generation of
            (
               "CLUSTER_19_13.bcg"
               |[RECV_18_19, SEND_19_18]|
               (
                  "MEDIUM_19_18.bcg"
                  |[RECV_19_18, SEND_18_19]|
                  "ACTIVITY_18.bcg"
               )
            );

"SUB_MEDIUM.bcg" =
   abstraction "TMP.bcg" of "MEDIUM_19_18.bcg";

% echo -n "checking if a bag medium with $N places is large enough: "

(* using the Evaluator model-checker of CADP, we check if SUB_MEDIUM.bcg
 * contains a sequence of $N consecutive "SEND_xx_yy" actions *)

% RES=‘bcg_open SUB_MEDIUM.bcg evaluator CHECK_$N.mcl | grep ’\<TRUE\>’‘
% if [ "$RES" = "" ]
% then
%    echo "yes"
%    break
% else
%    echo "no! (retrying with a larger bag medium)"
%    N=‘expr $N + 1‘
% fi

% done
```

Fig. 4. An excerpt of the generated SVL script.

Number of controllers	1	1	1	2
Number of containers	1	2	3	2
Total number of agents	2	3	4	4
Number of activities	13	21	29	34
Minimal size of activities (states)	7	7	7	7
Mean size of activities (states)	42	57	82	68
Maximal size of activities (states)	104	225	481	195
Number of media	18	30	42	36
Minimal size of media (states)	2	2	2	2
Mean size of media (states)	57	60	61	58
Maximal size of media (states)	111	111	111	111
Number of concurrent processes	31	51	71	70
Size of potential state space (states)	2.10^{24}	3.10^{41}	4.10^{68}	9.10^{68}
Size of largest generated LTS (states)	1,824	48,819	410,025	76,399
Size of generated LOTOS file (lines)	2,597	4,494	6,391	7,208
Size of generated SVL file (lines)	617	1,013	1,409	1,635
Number of intermediate files	221	316	503	519
Verification time	4 min 09	9 min 52	19 min 43	12 min 10

Fig. 5. Collected data for several configurations of the deployment protocol.

- Although the high number of concurrent processes could lead to a potentially huge state space (up to 9.10^{68} states if we estimate its size by multiplying the numbers of states of the minimized LTSs corresponding to all activities and media for a given configuration), compositional verification allows to keep the state space to a tractable size (below 10^6 states).
- Given the large number of intermediate files (several hundreds), these experiments would not have been possible without the SVL language and associated compiler.

The correctness of each protocol configuration was determined by checking in the corresponding global LTS the absence of `ERROR` messages (which denote either protocol design errors or implementation errors in the OCAML translator). When the LOTOS process corresponding to an activity is generated without its environment, all possible `ERROR` messages can be observed. However, when the process gets synchronized with all its children processes, there should not remain any `ERROR` message tagged with this process identity. For each configuration, we obtained the LTS of Figure 6, in which all communications are hidden but those made by the root activity of a controller. This LTS summarizes the service provided by the protocol to the end-user. The initial state has number 0. The two first transitions start the deployment by sending a start event and its acknowledgement. Then, either the user indicates that the deployment is not ready for activation, which will cause a failure notification from subactivities prevented from deploying components, or the user activates the deployment and receives either a notification that the deployment is successful or a failure indication.

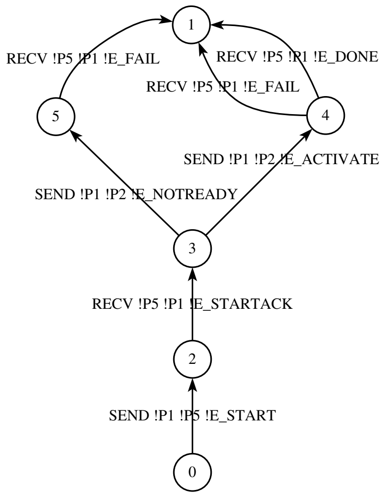


Fig. 6. Service provided by the root controller, as an LTS obtained by compositional verification using CADP.

6 Conclusion

Considering the numerous publications dealing with compositional verification and its expected benefits, it is high time to put this approach into practice in real-life case studies. Object-based distributed systems are an ideal target for this purpose, as they usually contain many components, either to model physically distributed entities or to represent concurrent activities taking place on a given execution site. This is the case with the SCALAGENT deployment protocol, whose architecture consists of a tree of distributed agents, each agent being itself organized as a tree of concurrent activities.

From the verification activities undertaken in the PARFUMS project, we can draw a number of conclusions.

The complexity of the SCALAGENT deployment protocol grows quickly as new components are added to the system. For instance, adding a new agent to deploy may start not less than 20 additional concurrent processes. For this reason, it seems that only compositional techniques have a chance to cope with the corresponding state explosion.

Because the SCALAGENT deployment protocol is implemented in JAVA, we could have tried to apply a software model checker (such as the JAVA PATHFINDER [21] or BANDERA [5]) directly on the JAVA source code. We did not choose such an approach because, to the best of our knowledge, these tools can only analyze programs running on a single JAVA virtual machine (JVM), whereas the SCALAGENT protocol is designed for multiple machines, each running a separate JVM. We also felt that a process algebra such as LOTOS, with its built-in concurrency and abstraction primitives, would provide better support for compositional modeling and verification. As a consequence, our verification efforts mostly addressed the higher level design (i.e., the reference specification of the protocol) rather than the implementation (i.e., the Java code) although an examination of the latter was sometimes needed.

Technically, the results of the verification effort are positive. Several ambiguities were found in the reference specification and the verification work exhibited an undocumented assumption (synchrony of communications) of crucial importance for a proper functioning of the protocol. The use of compositional verification allowed to check significantly complex finite configurations within a reasonable amount of time (at the moment, configurations with 70 concurrent processes can be verified in less than 20 minutes).

In modeling the ScalAgent deployment protocol, we chose to introduce many distributed buffers, instead of one central buffer. This avoids a bottleneck problem, which might prevent compositional verification from being applied [6]. We also took advantage of the "doubly nested" tree-like structure of the ScalAgent deployment protocol. Originally designed to ensure the scalability of the protocol when deploying software components on many machines, this tree-like structure also forms the skeleton of our compositional verification scenarios, in which Lts generation and minimization phases are incrementally performed from the leaves to the root of the trees.

Last but not least, a complex system such as the ScalAgent deployment protocol could not be analyzed in absence of mature verification tools. We found the Cadp toolbox robust enough for this challenge, but had to extend it in several ways. Two existing tools (Exp.Open and Projector) had to be entirely rewritten for performance reasons. A new tool, Bcg_Graph, was introduced for fast, automatic generation of communication buffers. The SVL scripting language was enriched to allow a wider form of parameterization. Interestingly, Svl scripts, originally to be written by human experts, are now automatically generated by the Ocaml translator. We observe here a situation in which new software layers are continuously added on top of existing ones, an integration trend that also occurs in other branches of computer science.

As regards future work, we foresee two directions. First, we are currently attacking larger configurations (90 and more concurrent processes) so as to discover the actual limits of compositional verification. Second, we seek to detect various livelock situations automatically using the Evaluator 3.0 model checker [18]; as the μ-calculus formulas needed to characterize livelocks may depend on the set of components defined in the Xolan architectural description, the Ocaml translator could be extended to generate these formulas automatically. This would reduce the risk of error and keep the verification fully transparent to the end-user.

Acknowledgements

The authors are grateful to Roland Balter, Luc Bellissart, and David Felliot, for sharing their knowledge of the ScalAgent deployment protocol. They would also like to thank the Parfums project manager, Laurent Coussedière, as well as David Champelovier, Radu Mateescu, and Wendelin Serwe for their useful comments about this article.

References

1. Gilles Barthe, Dilian Gurov, and Marieke Huisman. Compositional Verification of Secure Applet Interactions. In R.-D. Kutsche and H. Weber, editors, *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering FASE'02 (Grenoble, France)*, LNCS 2306, pages 15–32. 2002.
2. Frances Brazier, Frank Cornelissen, Rune Gustavsson, Catholijn M. Jonker, Olle Lindeberg, Bianca Polak, and Jan Treur. Compositional Design and Verification of a Multi-Agent System for One-to-Many Negotiation. In *Proceedings of the Third International Conference on Multi-Agent Systems ICMAS'98*. IEEE, 1998.
3. Ghassan Chehaibar, Hubert Garavel, Laurent Mounier, Nadia Tawbi, and Ferruccio Zulian. Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In R. Gotzhein and J. Bredereke, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'96 (Kaiserslautern, Germany)*, pages 435–450. IFIP, 1996. Full version available as INRIA Research Report RR-2958.
4. S. C. Cheung and J. Kramer. Checking Safety Properties Using Compositional Reachability Analysis. *ACM Transactions on Software Engineering and Methodology*, 8(1):49–78, January 1999.
5. James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering ICSE'2000 (Limerick Ireland)*, pages 439–448, June 2000.
6. Manuel Aguilar Cornejo, Hubert Garavel, Radu Mateescu, and Noël de Palma. Specification and Verification of a Dynamic Reconfiguration Protocol for Agent-Based Applications. In Aleksander Laurentowski, Jacek Kosinski, Zofia Mossurska, and Radoslaw Ruchala, editors, *Proceedings of the 3rd IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems DAIS'2001 (Krakow, Poland)*, pages 229–242. IFIP, 2001. Full version available as INRIA Research Report RR-4222.
7. Hubert Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In B. Steffen, editor, *Proceedings of the First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98 (Lisbon, Portugal)*, LNCS 1384, pages 68–84. 1998. Full version available as INRIA Research Report RR-3352.
8. Hubert Garavel and Frédéric Lang. SVL: a Scripting Language for Compositional Verification. In M. Kim, B. Chin, S. Kang, and D. Lee, editors, *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea)*, pages 377–392. IFIP, 2001. Full version available as INRIA Research Report RR-4223.
9. Hubert Garavel, Frédéric Lang, and Radu Mateescu. An Overview of CADP 2001. *European Association for Software Science and Technology (EASST) Newsletter*, 4:13–24, August 2002. Also available as INRIA Technical Report RT-0254.
10. D. Giannakopoulou, J. Kramer, and S. C. Cheung. Analysing the behaviour of distributed systems using TRACTA. *Journal of Automated Software Engineering, Special issue on Automated Analysis of Software*, 6(1):7–35, January 1999.
11. S. Graf, B. Steffen, and G. Lüttgen. Compositional Minimization of Finite State Systems using Interface Specifications. *Formal Aspects of Computation*, 8(5):607–616, September 1996.

12. Susanne Graf and Bernhard Steffen. Compositional Minimization of Finite State Systems. In R. P. Kurshan and E. M. Clarke, editors, *Proceedings of the 2nd Workshop on Computer-Aided Verification (Rutgers, New Jersey, USA)*, LNCS 531, pages 186–196, 1990.
13. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, 1989.
14. Guoping Jia and Susanne Graf. Verification Experiments on the MASCARA Protocol. In Matthew Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software SPIN'2001 (Toronto, Canada)*, LNCS 2057, pages 123–142, 2001.
15. Jean-Pierre Krimm and Laurent Mounier. Compositional State Space Generation from LOTOS Programs. In Ed Brinksma, editor, *Proceedings of TACAS'97 Tools and Algorithms for the Construction and Analysis of Systems (University of Twente, Enschede, The Netherlands)*, LNCS 1217, 1997. Extended version with proofs available as Research Report VERIMAG RR97-01.
16. Frédéric Lang. Compositional Verification using SVL Scripts. In Joost-Pieter Katoen and Perdita Stevens, editors, *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2002 (Grenoble, France)*, LNCS 2280, pages 465–469, 2002.
17. X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system (relase 3.06), documentation and user's manual, 2002. `http://caml.inria.fr/ocaml/htmlman/index.html`.
18. Radu Mateescu and Mihaela Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, March 2003.
19. Willem-Paul de Roever, Frank de Boer, Ulrich Hanneman, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification – Introduction to Compositional and Noncompositional Methods*, volume 54 of *Cambridge Tracts in Theoretical Computer Science*. 2001.
20. K. K. Sabnani, A. M. Lapone, and M. U. Uyar. An Algorithmic Procedure for Checking Safety Properties of Protocols. *IEEE Transactions on Communications*, 37(9):940–948, September 1989.
21. W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In Yves Ledru, editor, *Proceedings of the 15th IEEE International Conference on Automated Software Engineering ASE'2000 (Grenoble, France)*, pages 3–12, 2000.

Verification of Distributed Object-Based Systems[⋆]

Fernando L. Dotti[2], Luciana Foss[1],
Leila Ribeiro[1], and Osmar M. dos Santos[2]

[1] Instituto de Informática, Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
{lfoss,leila}@inf.ufrgs.br
[2] Faculdade de Informática, Pontifícia Universidade Católica do Rio Grande do Sul
Porto Alegre, Brazil
{fldotti,osantos}@inf.pucrs.br

Abstract. Distributed systems for open environments, like the Internet, are becoming more frequent and important. However, it is difficult to assure that such systems have the required functional properties. In this paper we use a visual formal specification language, called Object-Based Graph Grammars (OBGG), to specify asynchronous distributed systems. After discussing the main concepts of OBGG, we propose an approach for the verification of OBGG specifications using model checking. This approach consists on the translation of OBGG specifications into PROMELA (PROcess/PROtocol MEta LAnguage), which is the input language of the SPIN model checker. The approach we use for verification allows one to write properties based on the OBGG specification instead of on the generated PROMELA model.

1 Introduction

The development of distributed systems is considered a complex task. In particular, assuring the correctness of distributed systems is far from trivial if we consider the characteristics open systems, like: massive geographical distribution; high dynamics (appearance of new nodes and services); no global control; faults; lack of security; and high heterogeneity. It is therefore necessary to provide methods and tools for the development of distributed systems such that developers can have a higher degree of confidence in their solutions.

We have developed a formal specification language [6], called Object-Based Graph Grammars (OBGG), suitable for the specification of asynchronous distributed systems. Currently, models defined in this formal specification language can be analyzed through simulation [3] [4]. Moreover, starting from a defined model we can generate code for execution in a real environment, following a straightforward mapping from an OBGG specification to the Java programming

[⋆] This work is partially supported by HP Brasil - PUCRS agreement CASCO (24° TA.), ForMOS (FAPERGS/CNPq), PLATUS (CNPq), IQ-Mobile II (CNPq/CNR) and DACHIA (FAPERGS/IB-BMBF) Research Projects.

language [4]. In order to deal with open environments, we have worked on an approach to consider classical failure models still during the specification phase, allowing one to reason about a given model in the presence of a selected failure [7]. Investigations about the complexity of verifying properties of OBGG specifications considering message passing as the main operation were done in [12]. By using the methods and tools mentioned above we have defined a framework to assist the development of distributed systems. The innovative aspect of this framework is the use of the same formal specification language (OBGG) as the underlying unifying formalism [5].

In this paper we focus on the verification of object-based distributed systems. More specifically, we add to our framework the possibility of model checking distributed systems written according to the OBGG formalism. To achieve that, we propose a mapping from OBGG specifications to PROMELA (PROcess/PROtocol MEta LAnguage, which is the input language of the SPIN model checker), showing the semantic compatibility of the original OBGG specification with the PROMELA generated model. After that, we show how to specify properties using LTL (Linear Temporal Logic) over the OBGG specification. An important aspect is that the user does not need to know the generated PROMELA model to specify properties.

This paper is organized as follows: Section 2 presents the formal specification language OBGG together with an example (modeling the dining philosophers problem); Section 3 shortly introduces PROMELA; in Section 4 the translation from OBGG to a PROMELA model is presented together with a discussion of its semantic compatibility; in Section 5 we present our approach for the verification of OBGG models using SPIN and then conclude in Section 6.

2 The Specification Language OBGG

Graphs are a very natural means to explain complex situations on an intuitive level. Graph rules may complementary be used to capture the dynamical aspects of systems. The resulting notion of graph grammars generalizes Chomsky grammar from strings to graphs [8, 14]. The basic concepts behind the graph grammars specification formalism are:

- states are represented by *graphs*;
- possible state changes are modeled by *rules*, where the left- and right-hand sides are graphs; each rule may *delete, preserve and create* vertices and edges;
- a rule have *read access* to items that are preserved by this rule, and *write access* to items that are deleted/changed by this rule;
- for a rule to be *enabled*, a match must be found, that is, an image of the left-hand side of a rule must be found in the current state;
- an enabled rule may be *applied*, and this is done by removing from the current graph the elements that are deleted by the rule and inserting the ones created by this rule;
- two (or more) enabled rules are in *conflict* if their matches need write access to common items;

– many rules may be applied in *parallel*, as long as they do not have write access to the same items of the state (even the same rule may be applied in parallel with itself, using different matches).

Here we will use graph grammars as a specification formalism for concurrent systems. The construction of such systems will be done componentwise [13]: each component (called entity) is specified as a graph grammar; then, a model of the whole system is constructed by composing instances of the specified components (this model is itself a graph grammar). Instead of using general graph grammars for the specification of the components, we will use Object-Based Graph Grammars (OBGG) [6]. This choice has two advantages: on the practical side, the specifications are done in an object-based style that is quite familiar to most of the users, and therefore are easy to construct, understand and consequently use as a basis for implementation; on the theoretical side, the restrictions guarantee that the semantics is compositional, reduce the complexity of matching (allowing an efficient implementation of the simulation tool), as well as ease the analysis of the grammar. Basically, we impose restrictions on the kinds of graphs that are used and on the kind of behaviors that rules may specify.

Each graph in an OBGG is composed of instances of the vertices and edges shown in Figure 1. These vertices represent entities and elements of abstract data types. Elements of abstract data types are allowed as attributes of entities and/or parameters of messages (in the visual representation, such attributes are drawn inside the entity). Messages are modeled as (hyper)arcs which have one entity as target and as sources the message parameters (that may be references to other entities or values).

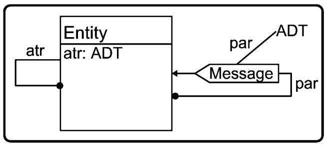


Fig. 1. Object-Based Type Graph.

For each entity, a graph containing information about all its attributes, relationships to other entities, and messages sent/received by this entity is built. This graph, called *type graph*, is an instantiation of the object-based type graph described above. All rules that describe the behavior of this entity may only refer to items defined in this type graph. Instances of entities are called *objects*.

A rule describes the reaction of objects to the receipt of a message. A rule of an OBGG must delete exactly one message (trigger of the rule), may create new messages to all objects involved in the rule, as well as change the values of attributes of the object to which the rule belongs. A rule shall not delete or create attributes, only change their values. At the right-side of a rule, new objects may appear (instances of entities can be dynamically created). Besides, a rule may have a condition, which is an equation over the attributes of its left- and right-hand sides. A rule can only be applied if this condition is true.

An OBGG consists of a type graph, an initial graph and a set of rules. The type graph is actually the description of the (graphical) types that will be used in this grammar (it specifies the kinds of objects, messages, attributes and parameters that are possible – like the structural part of a class description). The initial graph specifies the start state of the system. Within the specification of an entity, this state may be specified abstractly (for example, using variables instead of values, when desired), and will only become concrete when we build a model containing instances of all entities (objects) involved in this system. As described above, the rules specify how the instances of an entity will react to the messages they receive.

According to the graph grammars formalism, the computations of a graph grammar are based on applications of rules to graphs. Rules may be applied sequentially or in parallel. Each state of a computation of an OBGG is a graph that contains instances of entities (with concrete values for their attributes) and messages to be treated. In each execution state, several rules (of the same or different entities) may be enabled, and are therefore candidates for execution. Rule applications only have local effects on the state. However, there may be several rules competing to update the same portion of the state. To determine which set of rules will be applied, we need to choose a set of rules that is consistent, i. e., a set in which no two or more rules have write access to (delete) the same resources. Due to the restrictions imposed in OBGG, write-access conflicts can only occur among rules of the same entity. When such a conflict occurs, one of the rules is (non-deterministically) chosen to be applied. This semantics is implemented in the simulation tool PLATUS [3, 4].

We can describe this semantics as a labeled transition system (LTS) in which the states are the reachable graphs, and each transition represents a rule application, having as label the name of the rule that was applied. Actually, there are many possible choices for the labels of transitions, ranging from very simple ones, like just the name of the rule, to more complex ones, containing also the identity of the object and message involved in the rule application, and even names and values of attributes changed by this computation step. As these labels are the events that can be observed in the semantics, if we have richer labels, we will be able to describe more complex properties over the system represented by this transition system. However, the number of types of events (labels) has a direct impact in the size of the state space of the (translated PROMELA) system, and the risk of state explosion during verification is quite high if we have many different labels. The choice of just having rule names as labels is a trade off between expressiveness for describing properties and the limitations of verification tools.

2.1 The Dining Philosophers Problem

In this Section we model the dining philosophers problem using OBGG. The type graph, and rules for the objects that compose the specification are presented in Fig. 2. Using the same type graph and set of rules, we present two different models for this problem, given by two different initial graphs, describing a symmetric and an asymmetric solution. We use these solutions in Section 5 in order to illustrate our approach for the verification of properties.

Traditionally the dining philosophers problem is described in the following scenario. In a table there are N philosophers and N forks (a fork between every philosopher). The philosophers spend some time thinking, and from time to time a philosopher gets hungry. In order to eat a philosopher must, exclusively, acquire its left and right forks. After eating a philosopher release both left and right forks and starts thinking again.

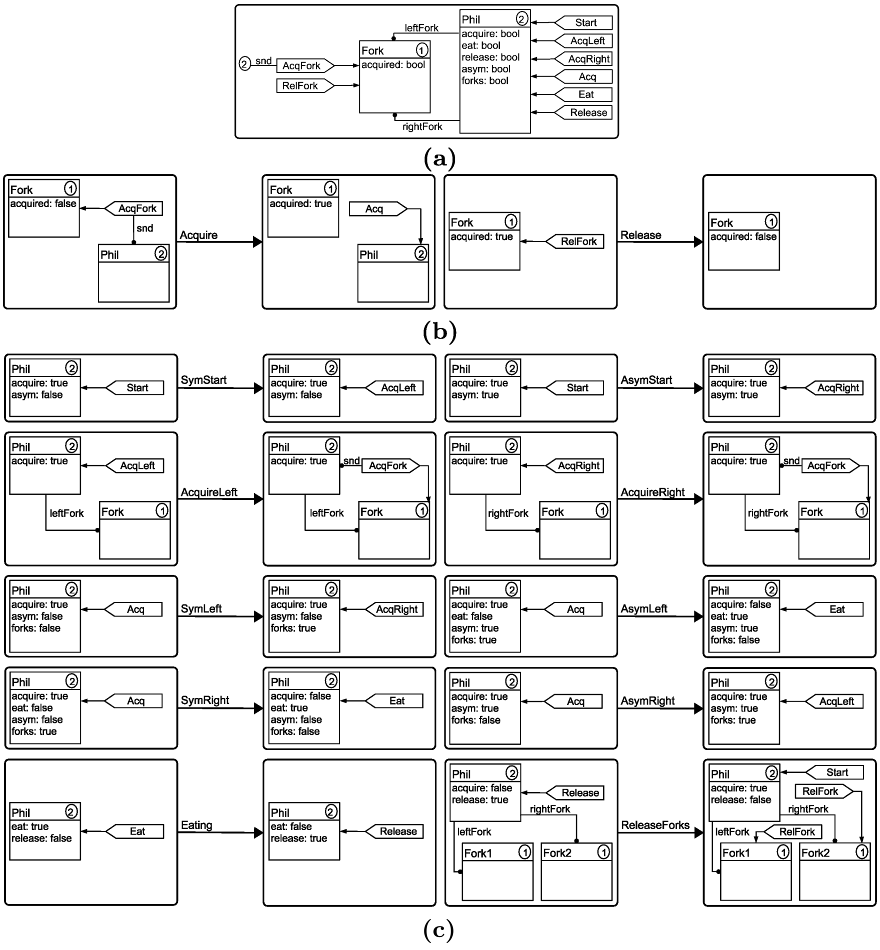


Fig. 2. Type Graph (a) and Rules of *Fork* (b) and *Phil* (c) Objects.

In OBGG we model the problem with two entities: *Fork* and *Phil.* The messages that objects of these entities can receive and their attributes are described in Fig. 2 (a)[1]. The *Fork* entity represents the forks and is composed of

[1] The numbers inside the circles are used to indicate the type of each entity. They are defined in the type graph and are used as a type information for the instances that appear in the rules and state graphs.

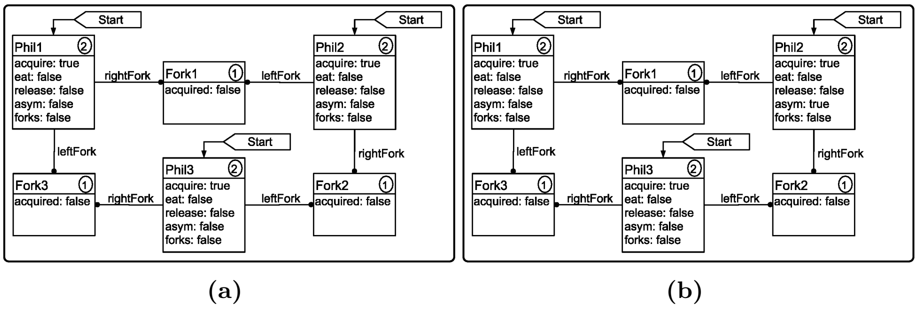


Fig. 3. Initial Graph for Symmetric (a) and Asymmetric (b) Solutions.

a boolean attribute (*acquired*) that determines if the fork is currently in use by a philosopher (*acquired* true) or not (*acquired* false). The *Phil* entity represents the philosophers and is composed of five boolean attributes: *acquire* (the philosopher is trying to acquire the forks), *eat* (the philosopher is eating), *release* (the philosopher is releasing its acquired forks), *asym* (indicates if the philosopher starts getting the left fork (false), or the right fork (true)), and *forks* (used to control the number of acquired forks). Each *Phil* object also have two references to *Fork* objects that are its left (*leftFork*) and right (*rightFork*) forks.

The rules for the *Fork* objects are shown in Fig. 2 (b), and the rules for the *Phil* objects are presented in Fig. 2 (c). The behavior of the specification is as follows. A philosopher starts execution, rules *SymStart* or *AsymStart*, trying to acquire its left (*asym* false) or right (*asym* true) forks (rules *AcquireLeft* and *AcquireRight*). If the philosopher can acquire the fork (rule *Acquire*), he tries to acquire the other fork (rules *SymLeft* or *AsymRight*). If the philosopher can acquire it too (rules *SymRight* or *AsymLeft*), he starts eating (rule *Eating*). After eating the philosopher release his forks and starts all over again.

In Fig. 3 (a) we show an initial graph for a symmetric solution of the problem. This solution is symmetric because all the philosopher has his *asym* attribute set to false, meaning that all of them will try to acquire the left fork first. Fig. 3 (b) illustrates an asymmetric solution for the problem. This solution is asymmetric because the philosopher *Phil2* has its *asym* attribute set to true, meaning that he will try to acquire the right fork first, differently from the other philosophers.

3 Process/Protocol MEta LAnguage

PROMELA [17] is a process based language, being used by the SPIN model checker [9] for the specification of models. In SPIN, from a PROMELA specification it is possible to define properties using LTL (Linear Temporal Logic) formulas, and verify if the formulas are true for a given specification.

The language has a *C-like* syntax and constructs for receiving and sending messages similar to the ones found in the specification language CSP (Communication Sequential Processes). Processes in PROMELA can be created statically or dynamically (*proctype* keyword). There is a special process, called *init*, used to initialize a specification. Processes can exchange information through message channels (*chan* keyword) or global variables (variables declared outside the scope of the processes). The message channels can be synchronous (the buffer of the message channel has 0 messages) or asynchronous (the buffer of the message channel can have N messages, being $N > 0$). Message channels are typed, in the sense that one has to explicitly declare the types of variables a channel might receive. Besides, PROMELA offers several functions used to check, for example, if a channel is not full (*nfull(channel)*), how many messages a channel has in its buffer (*len(channel)*), and others [17].

In PROMELA, non-determinism, i.e. the existence of more than one possible execution path, is modeled in condition (*if ... fi*) or repetition (*do ... od*) structures. The entries of condition and repetition structures are composed of guarded commands. Once the condition of a guarded command is not satisfied, the entry is blocked, possibly blocking the process that contains it. This blocking occurs until the condition is satisfied. In condition and repetition structures, non-determinism occurs when several entries have their conditions satisfied. In this case, one of the possible paths is chosen in a non-deterministic way. It is possible to define atomic structures (*atomic* { ... }) for a specification, i.e., a sequence of statements that must be executed without interleaving with the execution of statements of other processes. However, if there are guarded commands inside an atomic structure and they are not satisfied, the structure will lose its atomicity characteristic and will interleave its statements with other processes. We can define enumeration types in PROMELA (*mtype* keyword). The language provides a *goto* statement that enables a developer to jump into the body of a process. Finally, one can insert assertions in a PROMELA specification. An assertion statement evaluates an expression (*assert(expression)*) to true or false, each time the statement is executed. If the expression evaluates to false, an error is generated and the verification procedure stops.

We can describe the operational semantics of PROMELA by a labeled transition system (LTS). This semantics can be found in [15]. A state of a PROMELA program consists of a fragment of this program (its still unexecuted code), a function that relates each global variable (or channel) name with its value and a function that relates each channel identifier with its current value (length of channel buffer and values stored in it). Moreover, the state stores information about definitions of processes, about active processes and about which processes are currently executing atomic blocks. Each *proctype* statement defines a process. A process definition consists of the process body and a parameter function that defines the name and the type of its parameters. A process instantiation (active process) stores the following information: the body of the process and its current unexecuted code fragment, the values of all local variables (or channels) and a continuation stack that stores program fragments (used for do statements).

Each transition of a PROMELA LTS has as label some statement of the language. These transitions are defined by the SOS-rules of [15], which describe the behavior of each statement of PROMELA. The initial state for every LTS is $(\pi, G_\perp, C_\perp, pdef_\perp, act_\perp, \perp)$, where π is a program body; $G_\perp$, $C_\perp$, $pdef_\perp$ and $act_\perp$ are the mappings of global variables, channels, definitions and instantiations of process, respectively (all these functions are undefined for all values of their domains); and $\perp$ is a process identifier that is executing atomically.

4 Translation of OBGG into PROMELA

The translation of OBGG into PROMELA defines how the abstractions of OBGG are mapped to PROMELA, in a way that the semantics of OBGG is preserved. In Section 4.1 we present how the abstractions of object, message, rule and initial graph present in OBGG are translated to PROMELA, and in Section 4.2 we discuss how semantic compatibility is achieved. We do not show neither the concrete syntax of this mapping nor the proofs of compatibility due to space constraints.

4.1 Syntactical Mapping of OBGG into PROMELA

Objects and Messages. Objects in OBGG are translated into processes in PROMELA (we call such processes *object process(es)*). Attributes of an object are mapped to variables, passed as arguments in the definition of an *object process*. For verification purposes, attributes of OBGG objects are restricted to the types supported in PROMELA. A reference to an OBGG object is mapped to a PROMELA channel. Messages in OBGG are translated into messages in PROMELA. The receipt of messages is done through an asynchronous channel, called *object process channel*, that is also passed as an argument in the definition of an *object process*. The *object process channel* is typed according to: the name of a message (an *mtype* in PROMELA, composed of the name of all messages in the system type graph), and the parameters of all messages that an object can receive. The parameters of all messages that an object can receive become variables declared inside an *object process*. The dynamic creation of objects corresponds to the dynamic creation of processes and their associated channels in PROMELA.

Concurrency among objects is naturally preserved by the concurrency between *object processes*. Nevertheless, in OBGG it is possible to have intra-object concurrency. That is, when an object has several non-conflicting messages to be processed we may have the parallel reception of these messages. We face two main problems when translating this feature to PROMELA.

The first problem is due to the way messages are received in OBGG, i.e. messages are not stored in a specific place, they are simply connected to the object in the system state graph. Since we translate OBGG messages to messages sent through PROMELA channels, we require the user to set a buffer size for the *object process channel* of *object processes*. The problem occurs when a small size is set, introducing possible points of synchrony in the model (that do not exist in the original OBGG model because an object may receive an unbounded number of messages at each moment). We handled this problem by inserting assertions that, just before sending a message in the translated model, evaluate an expression to determine if the destination channel is not full. Thus, when verifying a model with a small buffer size, an error is generated when the *object process channel* is full, requiring the user to increase the buffer size.

The second problem is related to the non-deterministic reception of messages in OBGG. Because PROMELA channels work in a first-in first-out manner, we have to introduce a structure that non-deterministically receive messages to be processed. This is done by creating an internal buffer in every *object process* that is responsable for receiving a message in a non-deterministic way.

Thus, we define the generic behavior of an *object process* as follows: **(i)** wait for new messages in the *object process channel*; **(ii)** once new messages are received, send them to an internal buffer of the *object process*; **(iii)** non-deterministically choose a message from the internal buffer and try to apply a rule to process that message (see Rules below); **(iv. a)** if a message is processed and the *object process channel* is empty, return to **(iii)**; **(iv. b)** if no message is processed or the *object process channel* is not empty, return to **(i)**.

Rules. For an OBGG object there may exist several rules that are capable of handling the same message type. When receiving a message, one of the enabled rules that can handle the message is chosen in a non-deterministic way. We use a condition structure inside the *object process* to implement such abstraction. This condition structure has in its entries the necessary conditions to trigger the rules of the object (the match). Thus, an object with N rules will have N entries in this structure.

Initial Graph. The OBGG initial graph is composed of the instances of objects and the (initial) messages of the model. In our translation, the initial graph becomes an *init* process in PROMELA. This *init* process has three stages: **(i)** create the *object process(es) channel(s)* for objects that appear in the initial graph; **(ii)** execute the *object process(es)* defined in the initial graph, passing as arguments the values of its attributes and its *object process(es) channel(s)*; **(iii)** send defined (initial) messages using the *object process(es) channel(s).*

4.2 Semantic Compatibility

To assure that the translation preserves the OBGG semantics we have to prove (i) that every behavior in the OBGG-LTS can be found in the PROMELA-LTS of its translation and (ii) that no new behavior is added in the PROMELA-LTS. Due to a difference in the granularity of the LTSs, the latter proof can not be done, only a weaker version of it.

In order to carry out these proofs we translate the paths of the OBGG-LTS into paths of PROMELA-LTS **(i)** and vice-versa **(ii)**. For this, we must translate the states of the first into states of the second LTS, that is, we must find a correspondence between graphs and PROMELA states. We can always translate an OBGG state into a PROMELA state, but the opposite is not true. In the PROMELA-LTS of a program which results from the translation of an OBGG specification there are several states that do not correspond to any states of the OBGG-LTS. This is due to the fact that the treatment of messages in OBGG occurs atomically (in only one step), while in PROMELA this treatment occurs in several steps. Thus, in the PROMELA-LTS, there are states that represent the partial treatment of messages and these states do not have any corresponding state in OBGG-LTS. A PROMELA state that corresponds to some OBGG state will be called well-formed state. In a well-formed state, every instantiated process ($act(i) = (\pi_1, \pi_2, L, \varepsilon)$) must: not be more active ($\pi_1 = \varepsilon$); or not be executing ($\pi_1 = \pi_2$); or be ready to execute a labeled statement in the next step. The fist situation is the case of init process. The latter situation occurs when processes are waiting for messages or looking for enabled messages in the buffer. For a PROMELA-LTS state to correspond to a graph (OBGG-LTS state), it must have one active process (but not executing or with a labeled statement) for each object in the graph and one element in the channel or the buffer of an *object process* for each message in graph.

$pdef(Fork) = (\pi_3, f_{Fork})$,
$f_{Fork}(1) = (opc\_Fork, \text{CHAN})$
$f_{Fork}(2) = (atr\_acquire, \text{BOOL})$
$pdef(Phil) = (\pi_5, f_{Phil})$,
$f_{Phil}(1) = (opc\_Phil, \text{CHAN})$
$f_{Phil}(2) = (atr\_acquire, \text{BOOL})$
$f_{Phil}(3) = (atr\_eat, \text{BOOL})$
$f_{Phil}(4) = (atr\_release, \text{BOOL})$
$f_{Phil}(5) = (atr\_asym, \text{BOOL})$
$f_{Phil}(6) = (atr\_forks, \text{BOOL})$
$f_{Phil}(7) = (atr\_Fork\_leftFork, \text{CHAN})$
$f_{Phil}(8) = (atr\_Fork\_rightFork, \text{CHAN})$

$pdef(init) = (\pi_7, f_0)$

$G(event\_RuleName) = (\text{MTYPE}, 0)$

$C(0) = (\text{MTYPE}, Phil\_Start \cdot \varepsilon, 3)$
$C(1) = (\text{MTYPE}, Phil\_Start \cdot \varepsilon, 3)$
$C(2) = (\text{MTYPE}, Phil\_Start \cdot \varepsilon, 3)$
$C(3) = (\text{MTYPE} \times \text{CHAN}, \varepsilon, 3)$
$C(4) = (\text{MTYPE} \times \text{CHAN}, \varepsilon, 3)$
$C(5) = (\text{MTYPE} \times \text{CHAN}, \varepsilon, 3)$

(a)

$act(0) = (\varepsilon, \pi_7, L_1, \varepsilon)$
$L_1(Phil1) = (\text{CHAN}, 0)$
$L_1(Phil2) = (\text{CHAN}, 1)$
$L_1(Phil3) = (\text{CHAN}, 2)$
$L_1(Fork1) = (\text{CHAN}, 3)$
$L_1(Fork2) = (\text{CHAN}, 4)$
$L_1(Fork3) = (\text{CHAN}, 5)$
$act(1) = (\pi_5, \pi_5, L_2, \varepsilon)$
$L_2(opc\_Phil) = (\text{CHAN}, 0)$
$L_2(atr\_acquire) = (\text{BOOL}, 1)$
$L_2(atr\_eat) = (\text{BOOL}, 0)$
$L_2(atr\_release) = (\text{BOOL}, 0)$
$L_2(atr\_asym) = (\text{BOOL}, 0)$
$L_2(atr\_forks) = (\text{BOOL}, 0)$
$L_2(atr\_Fork\_leftFork) = (\text{CHAN}, 5)$
$L_2(atr\_Fork\_rightFork) = (\text{CHAN}, 3)$
$act(2) = (\pi_5, \pi_5, L_3, \varepsilon)$
$L_3(opc\_Phil) = (\text{CHAN}, 1)$
$L_3(atr\_acquire) = (\text{BOOL}, 1)$
$L_3(atr\_eat) = (\text{BOOL}, 0)$
$L_3(atr\_release) = (\text{BOOL}, 0)$
$L_3(atr\_asym) = (\text{BOOL}, 1)$
$L_3(atr\_Forks) = (\text{BOOL}, 0)$
$L_3(atr\_Fork\_leftFork) = (\text{CHAN}, 3)$
$L_3(atr\_Fork\_rightFork) = (\text{CHAN}, 4)$
$act(3) = (\pi_5, \pi_5, L_4, \varepsilon)$
$L_4(opc\_Phil) = (\text{CHAN}, 2)$
$L_4(atr\_acquire) = (\text{BOOL}, 1)$
$L_4(atr\_eat) = (\text{BOOL}, 0)$
$L_4(atr\_release) = (\text{BOOL}, 0)$
$L_4(atr\_asym) = (\text{BOOL}, 0)$
$L_4(atr\_Forks) = (\text{BOOL}, 0)$
$L_4(atr\_Fork\_leftFork) = (\text{CHAN}, 4)$
$L_4(atr\_Fork\_rightFork) = (\text{CHAN}, 5)$
$act(4) = (\pi_3, \pi_3, L_5, \varepsilon)$
$L_5(opc\_Fork) = (\text{CHAN}, 3)$
$L_5(atr\_acquire) = (\text{BOOL}, 0)$
$act(5) = (\pi_3, \pi_3, L_6, \varepsilon)$
$L_6(opc\_Fork) = (\text{CHAN}, 4)$
$L_6(atr\_acquire) = (\text{BOOL}, 0)$
$act(6) = (\pi_3, \pi_3, L_7, \varepsilon)$
$L_7(opc\_Fork) = (\text{CHAN}, 5)$
$L_7(atr\_acquire) = (\text{BOOL}, 0)$

(b)

Fig. 4. PROMELA State for OBGG Specification: Definitions, Global Variables and Channels (a) and Active Processes (b).

The initial state of an OBGG-LTS is the initial graph of the OBGG, which contains all objects and messages of the initial configuration of the system. The PROMELA-LTS initial state does not correspond to the OBGG-LTS initial state, because there is no running process in the initial state of the PROMELA-LTS. The state corresponding to the initial graph is the output state of the first transition labeled end_atomic (indicating that the *init* process (0) has ended and all processes corresponding to objects and messages present in the initial state of the system are running). The PROMELA state corresponding to the initial

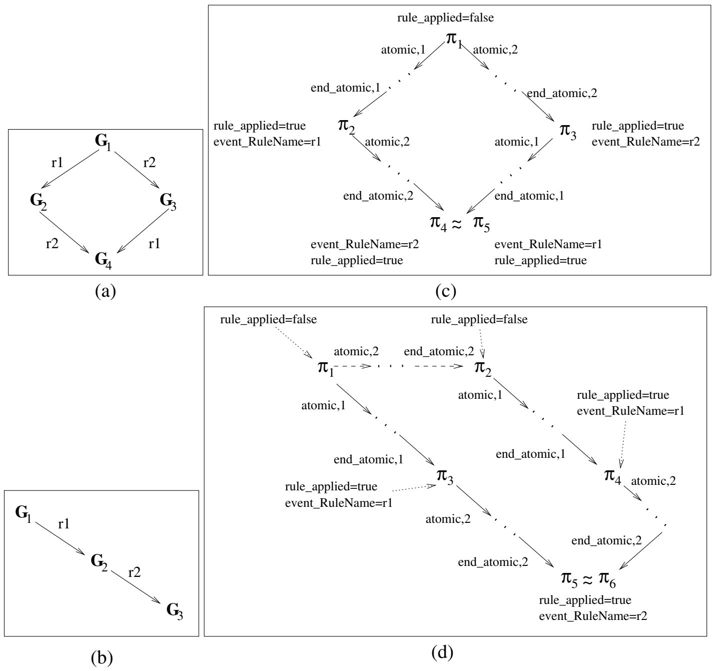


Fig. 5. OBGG-LTS's (a)(b) and PROMELA-LTS's (c)(d) of its Translation.

graph of Fig. 3 (b) is the state $ST = (\varepsilon, G, C, pdef, act, \perp)$, whose definitions are shown in Fig. 4 (π's definitions are omitted). The function *act* defines the instantiated and active processes. In Fig. 4, we can see that there is one active process for each object in the initial graph $(1-6)$. The functions L_n define the values of object attributes. For example, in the initial graph (Fig. 3 (b)) the *asym* attribute of *Phil*2 is true and in PROMELA-LTS state this same value can be seen in $L_3(atr\_asym) = (\text{BOOL}, 1)$. The messages and their parameters, present in the initial graph, are defined by function C, that defines the channel value of each object. All messages (and their parameters) sent to each object are in these channels.

The information described in the type graph of the OBGG can be found via the $pdef$ function, that associates to each type of object (process name) a process body modeling its behavior. We can also obtain the object's attributes, as well as their types through functions $f_{name\_process}$. The message types of system can be found in the *mtype* construct, that enumerates all types of messages and rule names of an entity. The message parameters and their types can also be obtained from the process body associated with each object by the function $pdef$. The first statements in the object process body are the declarations of parameters of all message types that the object can treat.

Besides translating the states we must translate the transitions of the LTS's. In the OBGG-LTS there is one transition for each rule application, but in the PROMELA-LTS there are several transitions that represent the same rule application. The behavior of an object process can result into two kinds of transition sequences, a *matching* sequence, corresponding to testing whether a rule can be applied (without applying the rule), and a *rule application* sequence testing and applying a rule. The matching sequence corresponds to situations when there is no match to apply a rule, and therefore it is only tested and not applied. As this transition sequence only makes tests, the part of the PROMELA state that corresponds to an OBGG state (graph) is not changed. Both sequences start with an "atomic" and end with an "end_atomic" transition. In the rule application sequence, in the final state of the "end_atomic" transition, the local variable *rule_applied* is *true*, whereas in the matching sequence the value of this variable is $false$.

Figure 5 shows examples of OBGG-LTSs and the PROMELA-LTS of their translations. In (a) and (c), we can apply $r1$ and $r2$ in parallel ($r1$ and $r2$ are independent) and, in (b) and (d), we can apply $r2$ only after $r1$ ($r2$ depends on $r1$). In (a) and (c), we can observe that the two rules can be applied in any order. The states π_4 and π_5 are equivalent, in this context, because they differ only in the value of the global variable *event_RuleName* and the local variable *rule_applied* of one *object process*. These variables do not interfere in the message or process states, and therefore the OBGG states corresponding to these two different PROMELA states are the same (isomorphic). In (d), the transitions represented by the dashed arrows correspond to idle transitions in the OBGG (because this corresponds to a matching sequence) and states π_5 and π_6 (and also π_3 and π_4) are equivalent, because the differences between them are not in message or object states.

The messages, in the states of PROMELA-LTS, do not have identifiers, so we are not capable of distinguishing, in a state, two messages with same type and parameters. In an OBGG, however, messages have unique identifiers. This means that the OBGG-LTS have a richer representation concerning the causality relationships among messages than the PROMELA-LTS. For each OBGG path there is one corresponding PROMELA path, but for the same PROMELA path there may be many different corresponding OBGG paths, each one representing a different causal relation between messages that is compatible with the PROMELA path. What is important to notice is that all these OBGG paths are in the transition system of the original OBGG-LTS, and therefore we are not adding new behavior with the translation of OBGG into PROMELA.

5 Verification of OBGG Specifications

The model checker SPIN is a state-based verification tool. The property formulas, written using LTL, are specified over the state of the system. More specifically, the developer must have global variables in the PROMELA model to specify and verify properties over it.

For the verification of OBGG specifications we have noticed that it is more natural for the developer to express properties about the application of rules rather than based on the state of specific objects. Moreover, if we use the state of an object (its attributes) in a property specification we would have to make available the values of such attributes through global variables. While this approach works for specifications with a static number of objects, it is not feasible for specifications that have dynamic creation of objects because we would need to dynamically create new global variables, a feature not supported by the tool.

Switching from the state-based approach presented above to an event-based approach, besides being more natural from the OBGG point of view (a rule application is seen as an event), we gain a more structured form to handle the verification of specifications that have dynamic creation of objects.

In order to specify properties using rule applications as events we have somehow to mark that a rule application is an event for verification. In our approach using events, every PROMELA model generated from the translation has a global variable *event_RuleName*. To generate events for the application of rules, we include the name of the rules that will become events into the *mtype* definition of this variable. Thus, when interesting events occur, i.e., the application of rules that are relevant for verification, they are written (using the *atomic* structure in PROMELA) into the *event_RuleName* global variable.

It is then possible to write LTL formulas about the occurrence of rules as being events, and these formulas need to inspect only the global variable *event_RuleName*. An event is the change of value of this global variable. For instance, we can define an event *eat* as being the change of value of the variable *event_RuleName* from *not Eating* to *Eating* (where *Eating* is the rule applied). We need to use the *next* temporal operator (X) to mark the change of value, for instance (! *Eating* && X *Eating*). The idea of specifying LTL formulas using events, and validating them with SPIN has been explored in [1].

As an example, we can define an LTL formula to specify that "it is always possible that some philosopher will eat", trying to prove that the specification is deadlock free. For that, we generate the events *Eating* (the philosopher is eating), *SymStart* (a philosopher is starting its execution), and *AsymStart* (a philosopher is starting its execution). This property is specified by the formula ([] <> *eat*), where ([]) is the always temporal operator and (<>) is the eventually temporal operator. We were able to verify this formula for the symmetric and asymmetric solutions of the dining philosophers problem. As expected, for the symmetric solution the formula does not hold, but the formula does hold for the asymmetric solution, where it used 530 Mb of memory, generated 6.21266e+06 states, and took 9 minutes running in an Intel Xeon 2.2 GHz Processor with a limit of 1 Gb of memory under SPIN.

Another property verified over the asymmetric and symmetric models concerned mutual exclusion. In a setting with up to three philosophers it is sufficient to prove that "no two philosophers might be in their critical sections (eating) at the same time". To prove such property we use the events *Eating* (the philosopher enters in the critical section) and *ReleaseForks* (the philosopher leaves the critical section). The formula ([] (*eat* && <> *rel*) $\rightarrow$ X (! *eat U rel*)) specifies this property, where (U) is the strong until temporal operator. Like for the event *eat* defined above, we define the event *rel* as being the change of value of the *event_RuleName* variable from *not ReleaseForks* to *ReleaseForks* (where *ReleaseForks* is the rule applied), leading to (! *ReleaseForks* && X *ReleaseForks*).

When verified, the formula for mutual exclusion was true and used 420 Mb of memory, generated 2.67997e+06 states, and took 2 minutes running in the same computer and configuration of the previous formula.

It is important to note that this approach implies in the use of the *next* operator in SPIN. In order to use the partial order reduction algorithm available, the SPIN model checker requires that, when using the *next* operator, the given formula is closed under stuttering. [1] has proposed a group of useful formula patterns for events that are closed under stuttering and can be directly applied in our work.

An important feature of our approach based on events is that the developer may write formulas looking only to the OBGG specification. It is not necessary to know the structure of the translated PROMELA model.

6 Final Remarks

In this article we have defined a translation from OBGG specification to PROMELA, and provided a way to verify properties over OBGG specifications based on events. We have also discussed the semantic compatibility between an OBGG specification and its corresponding PROMELA model.

The translation and integration between formal languages in order to use model checking tools is becoming a common practice, since many times it is easier (and more efficient) to reuse than to build a specific verification tool. Nevertheless, such translations involve detailed comparisons, especially at the semantic level. We can find in the literature various works focused on the verification of object-based/oriented distributed systems. The work proposed in [10] defines a visual and object-oriented language that can be mapped to the model checker SPIN. In [2] PROMELA is extended considering the actors concurrency model. [11] proposes a tool that tries to make available the automatic verification of UML models, this approach consists in the mapping of UML models to PROMELA. In [16] an integration of the formal specification language Object-Z with ASM (*Abstract State Machine*) was introduced, creating the OZ-ASM notation. After a series of translations, it is possible to verify OZ-ASM specifications using the SMV tool. In contrast to some of these works, in this article we discussed the semantic compatibility of our translation. Moreover, in our approach it is possible to specify the properties to be verified at the same level of abstraction of the specified OBGG model, a feature that is not present in most of the approaches that use translations.

For the verification of some aspects of a given specification, only the name of the applied rule as event name may be too few information. For instance, if we wish to prove the problem of fairness for the asymmetric solution of the dining philosophers problem, we would need to specify formulas about the individual behavior of a philosopher, like “for each philosopher i it is always possible that philosopher i will eat”. To support such level of detail we need to add more information to the event names. However, in doing so, we will have exponentially more states in the LTS to be verified. We are currently working on an approach to tackle this problem.

References

1. M. Chechik and D. O. Păun. Events in property patterns. In *5th and 6th Int. SPIN Workshops*, volume 1680 of *LNCS*, pages 154–167, Germany, 1999. Springer.
2. Seung Mo Cho et al. Applying model checking to concurrent object-oriented software. In *4th International Symposium on Autonomous Decentralized Systems*, pages 380–383, Japan, 1999. IEEE Computer Society Press.
3. B. Copstein, M. C. M0ra, and L. Ribeiro. An environment for formal modeling and simulation of control systems. In *33rd Annual Simulation Symposium*, pages 74–82, USA, 2000. IEEE Computer Society.
4. F. L. Dotti, L. M. Duarte, B. Copstein, and L. Ribeiro. Simulation of mobile applications. In *2002 Communication Networks and Distributed Systems Modeling and Simulation Conference*, pages 261–267, USA, 2002. The Society for Modeling and Simulation International.
5. F. L. Dotti, L. M. Duarte, F. A. Silva, and A. S. Andrade. A framework for supporting the development of correct mobile applications based on graph grammars. In *6th World Conference on Integrated Design & Process Technology*, pages 1–9, USA, 2002. Society for Design and Process Science.
6. F. L. Dotti and L. Ribeiro. Specification of mobile code systems using graph grammars. In *4th International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 177 of *IFIP Conference Proceedings*, pages 45–63, USA, 2000. Kluwer.
7. F. L. Dotti, O. M. Santos, and E. T. Rödel. On the use of formal specifications to analyze fault behaviors of distributed systems. In *1st Latin-American Symposium on Dependable Computing (accepted)*, LNCS, Germany, 2003. Springer.
8. H. Ehrig. Introduction to the algebraic theory of graph grammars. In *1st International Workshop on Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69, Germany, 1979. Springer.
9. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
10. S. Leue and G. Holzmann. v-Promela: a visual, object oriented language for SPIN. In *2nd International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 14–23, France, 1999. IEEE Computer Society Press.
11. Johan Lilius and Ivan Porres Paltor. vUML: a tool for verifying UML models. In *14th International Conference on Automated Software Engineering*, pages 255–258, USA, 1999. IEEE Computer Society Press.
12. A. B. Loreto, L. Ribeiro, and L. V. Toscani. Decidability and tractability of a problem in object-based graph grammars. In *17th IFIP World Computer Congress - Theoretical Computer Science*, volume 223 of *IFIP Conference Proceedings*, pages 396–408, Canada, 2002. Kluwer.
13. L. Ribeiro and B. Copstein. Compositional construction of simulation models using graph grammars. In *Application of Graph Transformations with Industrial Relevance (AGTIVE'99)*, volume 1779 of *LNCS*, pages 87–94, Germany, 2000. Springer.
14. G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation*, volume 1: Foundations, Singapore, 1997. World Scientific.
15. C. Weise. An incremental formal semantics for PROMELA. In *3rd SPIN Workshop*, The Netherlands, 1997.
16. Kirsten Winter and Roger Duke. Model checking object-Z using ASM. In *3rd International Conference on Integrated Formal Methods*, volume 2335 of *LNCS*, pages 165–184, Germany, 2002. Springer.
17. Promela language reference. http://spinroot.com/spin/Man/promela.html, 2003.

Design and Verification of Distributed Multi-media Systems

David Akehurst, John Derrick, and A. Gill Waters

University of Kent at Canterbury
{D.H.Akehurst,J.Derrick,A.G.Waters}@kent.ac.uk

Abstract. Performance analysis of computing systems, in particular distributed computing systems, is a complex process. Analysing the complex flows and interactions between a set of distributed processing nodes is a non-trivial task. The problem is exacerbated by the addition of continuous system functions that are time dependent, such as communication between components in the form of multimedia streams of video and audio data. Quality-of-Service (QoS) specifications define constraints on such communications and describe the required patterns of data transfer. By making use of these specifications as part of the performance analysis process it is possible to add significant confidence to predictions about the correct (required) operation of a distributed system. This paper presents a method for designing distributed multimedia systems, including the specification of QoS, using the ODP framework and UML and describes a technique for verifying the QoS specification against the designed functional behaviour of the system using Timed Automata.

1 Introduction

This paper demonstrates a specific approach to the design of distributed systems. The approach enables verification that Quality of Service (QoS) [15] specifications are met by the specified behavioural aspects of the design. The context which we aim to support is that of designers using the UML/OCL paradigm but requiring stronger verification than these would traditionally facilitate. This leads to an approach with design based around an adaptation of the UML supported by verification in a more formal setting provided by model checking timed automata.

The design includes the specification of both functional and non-functional aspects of behaviour. This is in the form of UML state diagrams for the functional behaviour and CQML statements to define the non-functional QoS of the system. The overall structure of the system and its components is specified using stereotyped UML class diagrams and a variation on UML object diagrams (snapshots) is used to define particular configurations of objects.

This approach to design enables us to verify that the environmental contract of an object (defined by the QoS specification) is met by the defined functional behaviour of that object. The verification is enabled by generating a network of timed automata (TA) from the design, which makes use of existing techniques for mapping UML state diagrams on to UPPAAL style timed automata templates. To map the QoS statements, we define timed automata templates that model the three QoS characteristics – latency, throughput and anchored jitter; these templates are instantiated using parameter values taken from the CQML statements.

A snapshot diagram (e.g. Fig. 1) is used to deduce a particular network of parallel automata constructed from the generated TA templates. This network can be model checked (using UPPAAL) to give feedback as to whether or not the functional behaviour verifiably conforms to the specified QoS. The event traces of UPPAAL can be used to construct a series of snapshots that illustrate the sequence of actions that lead to a problem. This enables the feedback to be in a form of the original design language rather than in the "lower level" analysis language.

The approach to design and verification described in this paper, builds on our work from a previous project [14], which address performance prediction from UML system designs and earlier work from this project [6], regarding the UML and specification of QoS. In particular we build on the work of [3], which describes a modelling language for the Computational Viewpoint that is an adaptation of the UML to enable design of distributed systems using the concepts defined in the RM-ODP. The work presented in this paper discusses an approach for mapping these high-level design specifications on to TA in order to verify the QoS aspects of the design. [6] deals with the static aspects of specifying QoS, the current work extends this by addressing the implication of the specified QoS on dynamic aspects of the system.

The rest of this paper is organised as follows. Section 2 illustrates our design approach using an example system. Section 3 describes the translation from specifications in our design languages into Timed Automata for the structure, behaviour and QoS of the system. Section 4 demonstrates the analysis technique incorporating the use of the UPPAAL model checker and illustrates how results can be fed back to the designer. Section 5 discusses some related work and concludes the paper.

2 An ODP Computational Specification

In this section we place our design approach in the context of the ODP framework and illustrate our approach to computational viewpoint design using an adaptation of the UML. Subsequently, in section 3, we will show how to translate these designs to TA in order to verify aspects of the design.

The computational viewpoint is concerned with the identification of distributable components (objects) and their interaction points (interfaces). The viewpoint addresses: the specification of the behaviour of identified objects; the specification of the signatures of the interfaces through which they interact; the specification of templates from which such components can be instantiated; and the specification of any constraints under which the objects must operate.

The current trend in software design communities is to make use of the UML as a specification language for system design. However, for the purpose of designing distributed systems (in particular within the context of the ODP computational viewpoint) the UML is deficient in a number of ways. To combat these deficiencies we have adapted the UML design language; this adaptation is presented and discussed in [3]. The description of the following example design highlights some of these adaptations.

In general, our design approach is to start with a snapshot of the system, to give an indication of the primary distributable components composing the system and the interfaces required to connect them. From the snapshot we identify and specify the computational object templates and interface signatures of the system. For each computational object we subsequently provide a behaviour specification. Finally we specify environment contracts for each computational object in the form of QoS constraints. As our illustrative example we take the specification of a lip synchronisation system, based on the specification presented in [4].

2.1 System Snapshot

A key aspect of a computational viewpoint specification is the decomposition of the system into distributable *objects* that interact at *interface*s. An object, which may be a composition of two or more other objects, is a unit of distribution and management that encapsulates behaviour [16]. To differentiate from the UML concept of an object, we shall use the term *computational object.*

Fig. 1 depicts a computational viewpoint snapshot of an example mobile videophone, including a lip synchronisation component. There are three aspects of the example system – transmitter device, channel, receiver device. The transmitter device's *camera* emits video frames to the *videoBinding* across the bound *transVideo* interfaces. The receiver device's v*idWindow* receives the video frames from the *Video-Binding* at the bound *recVideo* interfaces. On arrival of each video frame a signal is output to the *syncController* at the bound *vidReady* interfaces. Similarly the audio packets are emitted from the *microphone* computational object to the *AudioBinding* at the *transAudio* interfaces; they arrive across the binding at the speaker computational object and *recAudio* interfaces, where signals are emitted to the *syncController* at the *audReady* interfaces.

Transmission of the audio and video data via different channels applies a different and variable time delay to the media streams. As a result the computational object *syncController* is used to adjust the playback rate of the media streams to produce synchronised presentation. The *syncController* indicates when to display each video frame by emitting a signal at its *vidCtrl* interface, which is received by the *vidWindow*. Finally, the s*yncController* has an interface *appControl* that is used to reset the synchronisation algorithm.

A UML object diagram could have been used to illustrate the system configuration, but the notation would not effectively distinguish between computational objects, interfaces and bindings. To rectify this, we define an alternative notation which does distinguish between these concepts, enabling the presentation of computational viewpoint snapshots in a clearer fashion. Using the UML object diagram notation, the components would all appear as boxes, however, we use different syntactic representations for each category of component. This syntactic representation is linked to UML descriptions via a metamodel which effectively defines a transformation between the two representations [3].

In the adapted notation, circles depict computational objects. Binding objects are distinguished from computational objects by illustrating them as elongated circles.

Interfaces are illustrated using 'T' shapes, attached to a circle to indicate that the object (depicted by the circle) offers that particular interface. The role of the interface (producer/consumer, initiator/responder or client/server) is indicated by the direction and style of an arrow placed near the interface. Bound interfaces are either connected via an irregularly dashed line (e.g. *vidCtrl*) or placed head to head (all other bound interfaces in this snapshot – *transVideo*, *recVideo*, *transAudio*, *recAudio*, *vidReady*, *audReady* and *appControl*).

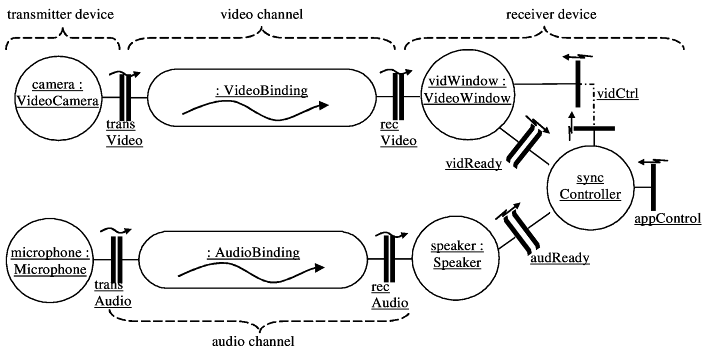


Fig. 1. Computational Viewpoint snapshot illustrating the Lip Synchronisation system

The identification policy for objects and interfaces is similar to the approach used in UML object diagrams, computational objects and interfaces are identified by either or both of an 'instance name' and a 'template name' separated by a colon and underlined. Where bound interfaces are close together we omit naming both interfaces separately and distinguish between them using their role. As with the UML object diagram notation, where appropriate, we need not give both template and instance name. In this snapshot the two bindings (for audio and video) are labelled with only the template name, the interfaces are labelled with only an instance name, as is *syncController*; other objects are labelled with both template and instance name. Object templates and interface signatures are specified in detail using a notation based on UML class diagrams as described in the next subsection.

2.2 Template and Signature Specifications

The snapshot discussed in the previous subsection indicates the kinds of component needed in order to build the system. The next step is to fully specify those components in order to obtain reusable and detailed definitions of the aggregated parts of the system. From a computational viewpoint, the necessary specifications include the definition of computational object templates, interface signatures, and the relationships between them.

The UML concept of class has a similarity to the ODP notion of template (and signature), thus, we define stereotypes of the UML class concept for specifying components based on the ODP concepts of: computational viewpoint object templates; stream, operational and signal binding objects; reactive objects; and stream, operational and signal interface signatures. This gives us a language and notation suitable for defining the computational viewpoint of an ODP system, which is (hopefully) familiar to UML designers; easily used; and provided with tool support from many standard UML tools.

Fig. 2 defines a template diagram for the instances specified in the computational snapshot shown in Fig. 1. Both object template and interface signatures are depicted using the notation for UML classes, distinguished using stereotype labels. To aid the distinction, computational object and binding object templates are shaded, whereas interface signatures are not. The stereotype of interface signatures also distinguishes between operations, stream and signal signatures.

The relationship between an object template and the interfaces that its instances may offer is specified using stereotyped UML associations. The stereotype of the association defines the role in which the object may offer instances of the interface signature; the association end name gives a navigation name for the object to refer to the interface. Each interface instance may be offered by only one object; hence the object end of the association is defined to be an aggregation (using a black diamond).

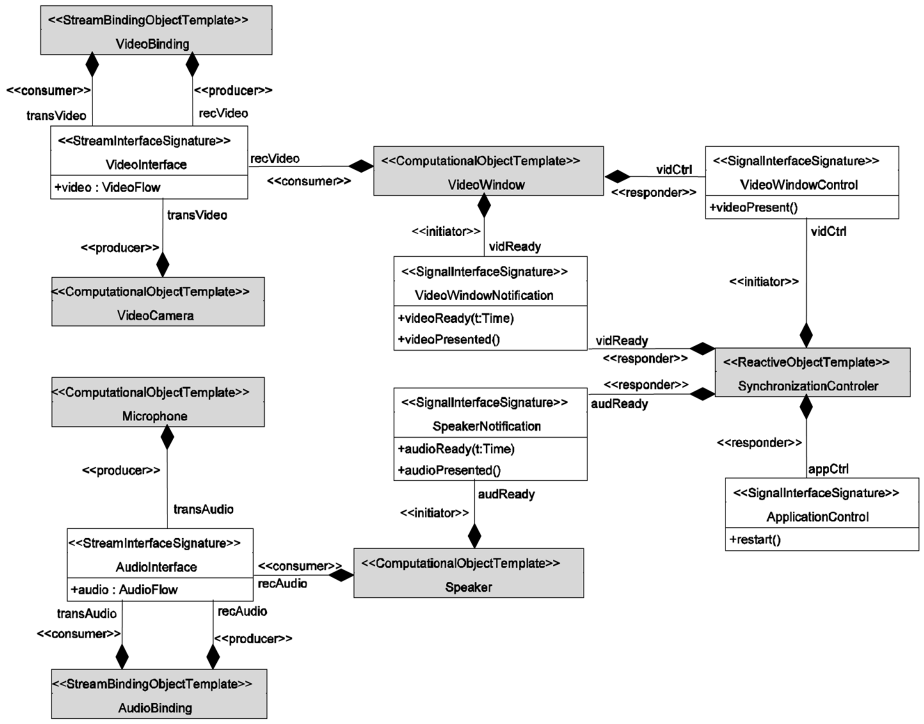


Fig. 2. Template Diagram for the Lip Synchronisation system

Given this specification, the objects and interfaces from the snapshot diagram are related to the appropriate template or signature. In our example, the snapshot already shows the relationship for all of the objects except the *syncController*, which is instantiated from the *SynchronisationController* template; the interfaces are instantiated from the defined signatures with a similar name. The snapshot diagram could be refined to show these relationships but we do not illustrate the refinement here.

2.3 Behaviour

After defining the object templates and the interfaces they may support, it is necessary to define the behaviour of the objects and the interactions that occur across the interfaces. This subsection firstly describes how UML State Diagrams can be adapted to specifying behaviour within the computational viewpoint and subsequently illustrates the technique by defining the behaviour of computational objects instantiated from the *SynchronisationController* template (i.e. *syncController*). It is assumed that the reader is familiar with standard State Diagrams (or *Statecharts* [10]).

Normally a State Diagram is associated with a UML class; it is used to specify the behaviour of objects instantiated from that class. Events that affect the state of the object and actions caused by the object are related to interactions (i.e. operations) defined on the class and other classes it is associated with.

In our model for design, interactions are defined on interfaces attached to an object, thus we cannot directly reference those interactions from the object. In addition, a reference to the interaction is not sufficient (on its own) to identify the cause of the event; an interface signature (and hence the interactions defined in it) may be instantiated multiple times for a single object.

In order to use State Diagrams in the context of our adapted design language, we associate a State Diagram with a particular computational object template; however, we have to alter the interpretation of the text specified on transitions.

Normally a transition is labelled with a string that has the following general format:

```
<event-signature> '[' <guard > ']' '/'<action-expressions>
```

Where, for standard State Diagrams, *event-signature* describes an event with its arguments; the *guard* condition is a Boolean expression written in terms of parameters of the triggering event and attributes of the object. The *action-expressions* are executed if and when the transition fires. Typically an action expression alters the local state of an object or causes another event to be fired, possibly by sending a message to another object.

Our adaptation requires the *event-signature* to identify the interface from which the interaction causing the event is received, along with the identifier for the interaction; *guard* conditions are interpreted in the standard manner; and *action-expressions* either:

1. Alter the local state of the object;
2. Instantiate interfaces to be offered by the object; or
3. Cause interactions at a specified interface; by identification of an interface and an interaction available at that interface.

This alteration to the text label of a transition enables State Diagrams to be used in the context of our adaptation to the UML design language.

Synchronisation Controller

The Synchronisation Controller objects have a complex specification. For a full description of the algorithm and the way in which it works refer to [4], it is not the purpose of this paper to describe a synchronisation algorithm, we simply use it as an example. A State Diagram (Fig. 3) is presented that specifies the behaviour.

Note that events are identified by an interface name and name from the signature of that interface, e.g. `appCtrl.restart`. Additionally, signals are sent within an action expression by identifying an interface name and a name from the signature of that interface, e.g. `vidCtrl.videoPresent`.

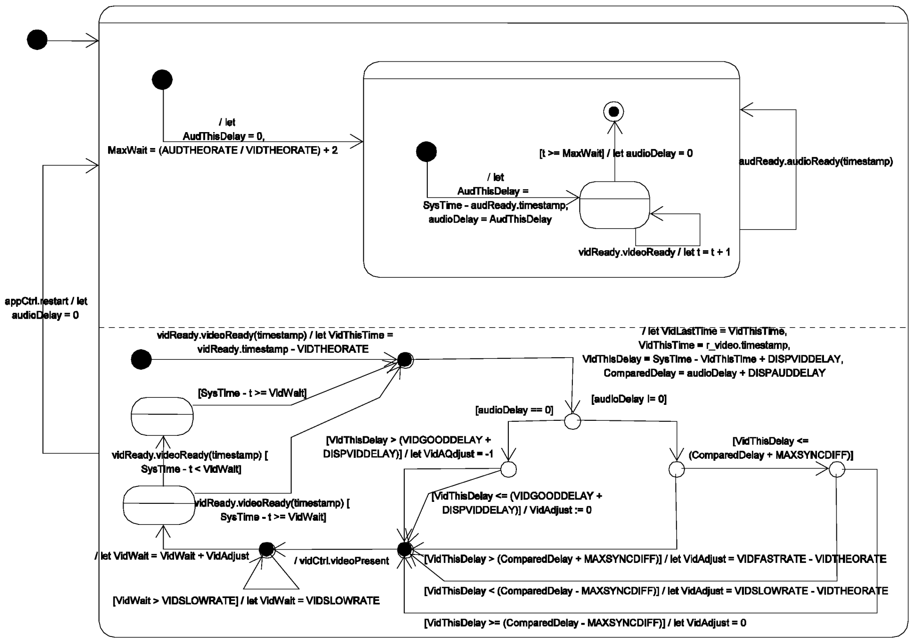

/ let AudThisDelay = 0, MaxWait = (AUDTHEORATE / VIDTHEORATE) + 2
[t >= MaxWait] / let audioDelay = 0
audReady.audioReady(timestamp)
/ let AudThisDelay = SysTime - audReady.timestamp, audioDelay = AudThisDelay
vidReady.videoReady / let t = t + 1
appCtrl.restart / let audioDelay = 0
vidReady.videoReady(timestamp) / let VidThisTime = vidReady.timestamp - VIDTHEORATE
/ let VidLastTime = VidThisTime, VidThisTime = r_video.timestamp, VidThisDelay = SysTime - VidThisTime + DISPVIDDELAY, ComparedDelay = audioDelay + DISPAUDDELAY
[SysTime - t >= VidWait]
[audioDelay == 0]
[audioDelay != 0]
[VidThisDelay > (VIDGOODDELAY + DISPVIDDELAY)] / let VidAQdjust = -1
[VidThisDelay <= (ComparedDelay + MAXSYNCDIFF)]
vidReady.videoReady(timestamp) [SysTime - t < VidWait]
[VidThisDelay <= (VIDGOODDELAY + DISPVIDDELAY)] / VidAdjust := 0
vidReady.videoReady(timestamp) [SysTime - t >= VidWait]
/ let VidWait = VidWait + VidAdjust
/ vidCtrl.videoPresent
[VidThisDelay > (ComparedDelay + MAXSYNCDIFF)] / let VidAdjust = VIDFASTRATE - VIDTHEORATE
[VidThisDelay < (ComparedDelay - MAXSYNCDIFF)] / let VidAdjust = VIDSLOWRATE - VIDTHEORATE
[VidWait > VIDSLOWRATE] / let VidWait = VIDSLOWRATE
[VidThisDelay >= (ComparedDelay - MAXSYNCDIFF)] / let VidAdjust = 0

Fig. 3. State Diagram specification of the Synchronisation Controller's behaviour

2.4 Environment Contracts – QoS Specification

The previous subsections have defined the structure, templates and functional behaviour of the system. We now address the specification of non-functional aspects of the system by defining some QoS constraints.

The ODP standard defines the concept of an environment contract, which it uses to describe QoS constraints. This is a contract between an object and its environment, i.e. all other objects with which it interacts. As interactions occur across interfaces, environment contacts for an object generally involve one or more interfaces. A QoS constraint involves two parts:

1. Requirements of the object by the environment, known as obligations; and
2. Requirements of the environment by the object, known as expectations.

The relationship between these two parts states that "provided the expectations are met (by the environment) the obligations will be met (by the object)". The expectations and obligations are expressions that constrain the QoS characteristics of interactions with the object.

There are many possible QoS characteristics that could be constrained. For the purpose of this paper we look at three stream and time related characteristics – latency, anchored jitter and throughput. *Latency* is the amount of time between two events (e.g. time between sending a frame and receiving it); *throughput* is the rate of occurrence of events (e.g. the rate of flow of frames); and *anchored jitter* is a variation in nominal throughput (e.g. variation in rate of flow of frames).

There is currently no clear contender for a most commonly used (de facto) QoS specification language. One that we have found to be most suited to our design approach, partly due to its association with the Object Constraint Language (OCL) and UML, is the Component Quality Modelling Language (CQML) [1]. CQML is a lexical language for QoS specification and has been developed to explicitly include as many features as possible. We have found the language to be expressive, very useable and easily adapted to integrate with our other UML based languages used within the ODP framework. The CQML semantics require that the interaction constrained by a characteristic must facilitate access to a historical sequence of events. The definition of *quality characteristic*s, such as latency, throughput and anchored jitter, is expressed in terms of the history of events (we do not provide the definition of these characteristics in this paper). Constraints regarding particular characteristics are formed in CQML by specifying *quality statement*s; these are grouped to form QoS specifications on particular components as *QoS Profile*s. A QoS profile includes statements for both e*xpectations* and *obligations*; each *expectation* or *obligation* is an expression referring to one or more quality statements. The quality statements enable reuse of QoS specifications across multiple QoS profiles.

A quality statement contains the conjunction of a number of sub expressions that constrain a variety of quality characteristics. Each quality characteristic is defined by an OCL expression that (in the case of latency, anchored jitter and throughput) references a particular interaction. To enable quality characteristics to be generalised and reused, they can be defined with specific parameters.

Given a set of pre-defined quality characteristics (*throughput*, *anchoredJitter* and *latency*) the QoS specifications associated with the Template diagram of Fig. 2 can be defined. We illustrate this (below) using the *VideoBinding* component. QoS specifications for other parts of the system can be found in [2].

Use of CQML, in the context of our model for design, means that QoS profiles are associated with computational objects (or their templates) and the constrained interactions are identified by reference to an interface and an appropriate interaction. The requirement to provide event sequences is met by the 'Event Notification Function', defined in the RM-ODP [16], which also requires event histories to be made available. We alter the standard CQML notation slightly, changing the keywords *profile*, *uses* and *provides* into *QoSProfile*, *exp:* and *obl:* as we find this allows the expressions to be more easily understood in the context of the ODP framework terminology; we also do not require profiles to be named.

VideoBinding

The *VideoBinding* from camera to video window is specified to provide a through frame rate of no less than 25 fps with a latency of between 40 and 60 milliseconds (ms) so long as it receives an input frame rate of no less than 25 fps. This is expressed in CQML as follows:

```
QosProfile for VideoBinding {
  exp: quality {
   throughput(1000,transVideo.video)>=25; };
  obl: quality {
   throughput(1000,recVideo.video)>=25;
   latency(transVideo.video,recVideo.video).maximum=60;
   latency(transVideo.video,recVideo.video).minimum=40; }; }
```

The above *QoS Profile*, defined for the *VideoBinding* template, defines one *expectation*, that there should be at least 25 events received every second (1000 ms) at the '*video*' *VidowFlow* part of the consumer interface *transVideo*. It also defines that the binding is *obliged* to provide at least 25 frames every second (fps) from the *VideoFlow* (named '*video*') part of the *VideoInterface* signature (*recVideo*) supported by the binding in the role of a *producer*. Additionally there are constraints between consumer and producer *VideoFlows* that specify the maximum and minimum latency that should occur for a frame passing through the binding.

The particular *VideoFlow* interactions, on which the constraints are placed, are navigated to using the association end names of the associations relating object templates to interface signatures.

3 Translation to Timed Automata

To verify that the constraints defined by the QoS specifications are met by the functional behaviour, for a particular configuration of system components, we create a network of parallel timed automata. It is intended that the TA network be automatically generated by transforming the information contained in the design specification. There are three aspects to this transformation:

1. State Diagrams are mapped to hierarchical timed automata [8] which can be flattened for input to UPPAAL using the method described in [8].
2. The parameters of the QoS constraints are used as arguments to TA templates that model the appropriate QoS characteristic.
3. A snapshot configuration diagram is used to appropriately connect the set of TAs, produced by the first two steps, into a single network.

This gives a network of TA that model the whole system, plus the environment in which the system is supposed to execute. There are three different sets of automata arising from the transformation:

1. `Func` - Automata modelling the functional behaviour of objects.
2. `Obl` - Automata modelling the obligation QoS statements, which observe the outputs of objects.
3. `Exp` - Automata modelling the expectation QoS statements, which provide inputs to objects from the environment.

The behaviour of the original specification (i.e. the one containing the functional description together with CQML statements of QoS) is now represented by the parallel composition of these three sets of automata:

```
Func || Obl || Exp
```

This TA network is entered into the UPPAAL model checking tool [13], which is subsequently used to check for deadlock in the system. Formally, there are six situations which could cause deadlock: internal deadlock in an automaton from any of the three sets; and deadlock due to a missed synchronisation between a pair of automata each from any pair of the three sets:

a) Deadlock in a functional behaviour automaton (Func): occurs if the functional behaviour is badly designed and causes a deadlock.

b) Deadlock in one of the QoS obligation automata (Obl): indicates that the functional behaviour does not meet the QoS obligation; occurs if the automaton has entered a state that indicates a QoS violation. The QoS obligation automata are designed with specific deadlock states to indicate that the QoS constraints they represent have been violated.
c) Deadlock in one of the QoS expectation automata (Exp): will never occur.
d) Synchronisation deadlock between automata from Func and Obl: will never occur, the Obl automata will enter a specific deadlock state if they can't synchronise on an output event from a Func automata.
e) Synchronisation deadlock between automata from Exp and Func: indicates that the functional behaviour is expecting a different QoS to that provided by the automata from Exp.
f) Synchronisation deadlock between automata from Exp and Obl: will never occur, automata from these sets never synchronise on common events.

In short, a deadlock is due either to an internal problem of the functional behaviour of a computational object; or due to the computational object not interacting with its environment in the manner specified by the obliged or expected QoS.

The position of the deadlock can be fed back to the designer (see example below), indicating which QoS constraint has not been met by the system, or that the functional behaviour has deadlocked. This is illustrated in section 4 by analysis of our example system. Before we discuss the analysis, the following subsections describe the manner in which we have modelled the three QoS characteristics – latency, throughput and jitter – using Timed Automata. These automata are instantiated by giving values to their parameters, taken from the CQML specifications.

3.1 Throughput and Anchored Jitter

The characteristic of throughput is easily modelled as a continuous source of events; however, due to the properties of anchored jitter, it is unnecessary to separately model the two characteristics and we can define a single TA that models both. Anchored jitter is defined as a variation in throughput; the variation does not change the overall throughput rate (this would be non-anchored jitter), it simply means that given the expected time for arrival of an event, the event might arrive one or other side of this time. The expected arrival times of following events are not affected by the actual arrival time of events (as is the case with non-anchored jitter). Thus, the specification of anchored jitter is tied to the specification of a particular throughput (which gives the expected arrival times) and hence it is more efficient to model the two characteristics as a single automaton template.

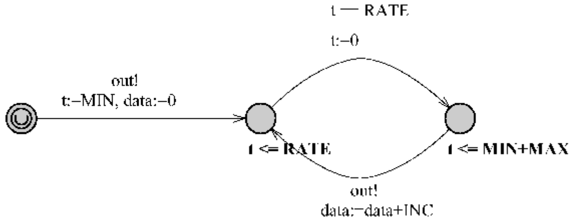


Fig. 4. Expectation Anchored Jitter and Throughput

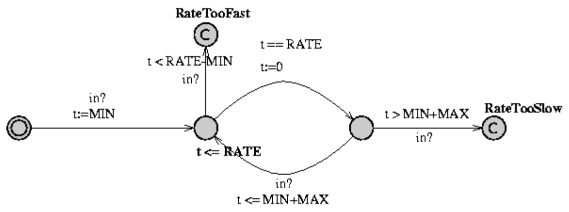


Fig. 5. Obligation Anchored Jitter and Throughput

A QoS expectation relating to throughput and anchored jitter can be encoded as the TA illustrated in Fig. 4. The automaton offers output events at a regular rate with an inter-event gap of 'RATE' (inverse of throughput) with an anchored jitter of between 'MAX' and 'MIN' above and below the value 'RATE'. The variable 'data' models a parameter value carried by the output event, for example a frame count, or as in the case of the lip sync example can hold a timestamp for the time the event (frame or packet) is created.

A pair of CQML quality statements defining the throughput and anchored jitter at a particular interface map to the parameter values of this automaton as follows:

For the quality statements:

```
quality {
  throughput >= X;
  anchoredJitter <= Y; }
```

The parameters RATE, MIN and MAX would be defined as:

```
RATE = 1 / X
MIN = MAX = Y
```

Note: The automaton models throughput using a fixed (through jittery) inter-arrival rate. As such it does not truly model the throughput characteristic used in the QoS specification of the example, which specifies a constraint on the total number of frames arriving within a specified time (i.e. 1000 milliseconds). Strictly speaking this TA is modelling the quality statement:

```
quality {
  minimum_inter_arrival_time = 1/X;
  anchoredJitter <= Y; }
```

Similarly, the TA shown in Fig. 5 assures a QoS obligation involving throughput and jitter. Note the important difference between the obliged and expected versions of this automaton. The obliged version is acting as an observer of offered outputs and does not force an output to occur.

The automaton will deadlock in state 'RateTooFast' if an output is offered too early; i.e. before time 'RATE-MIN' after the last output. Similarly it will deadlock in state 'RateTooSlow' if an output is offered too late; i.e. after time 'RATE+MAX' after the last output.

3.2 Latency

The QoS characteristic latency is defined as "the time that elapses between a *stimulus* and the *response* to it", i.e. latency is the relationship between the time one event occurs and the time that an associated second event occurs. To model this using timed automaton, it is necessary to be able to record the entry time for every stimulus event that is passing through the component by a clock. We can use each such clock to measure the time between stimulus and response. Since, the UPPAAL model of TA can only accept finite number of clocks, we must calculate how many are needed. Given only a latency specification, it is not possible to determine how many clocks are needed (i.e. how many events to track at any one time) and it would seem that a language allowing only a fixed number of clocks would not enable us to model this characteristic. However, no event will be generated (and need to be tracked) if the rate of generation of stimulus events is zero, i.e. without a throughput there is no need to check for latency. Therefore, if we have a defined throughput, or at least a defined minimum inter arrival time, it is possible to calculate how many clocks are required.

We calculate this value, called SIZE, as follows:

```
SIZE = latency * minimum inter arrival time
```

To model this as a Timed Automaton, we use a circular buffer of SIZE number of clocks to record the stimulus event times. The next available clock is reset when a stimulus event occurs and the time value of the clock is compared with the required latency when the corresponding response event occurs. If an attempt is made to reuse a clock before it has been checked (by response event occurrence) then essentially the buffer is too small. However, assuming that events do not arrive quicker than the minimum arrival rate, if the latency we are checking against is met by every event, the above calculation of SIZE ensures that the buffer is not too small; hence, buffer overflow indicates that latency has not been met. If, at any point in time the number of stimulus events passes SIZE, then the actual latency of some preceding event must have exceeded the value of latency against which we are checking.

The automaton template in Fig. 6 models this; the variable *t* is an array of clocks, *v* is an array of corresponding (frame or packet) identifiers, *eid* gives an index into arrays based on the identity of the arriving frame/packet, *latency* is a constant holding the value of latency to check, *start* and *check* are the stimulus and response channels and *st_id* and *ch_id* carry the identity of the frame/packet represented by the stimulus and response channels.

4 Performance Analysis

This section first describes the result of verifying our illustrative example, followed by a description of a technique to present back to a designer, in the context of the original design language, a trace originally produced by UPPAAL in terms of the generated automata.

4.1 Verification of the Example

As discussed above, the UPPAAL verifier is used to check the model against a query stating that there are no deadlocks for all time: `A[] not deadlock`

For our example, the UPPAAL verifier indicates that there is a deadlock; its cause is that the QoS specified for the video window is not met by its behaviour, i.e. situation (b) of those described in section 3. UPPAAL provides a trace of Automata events that lead to the deadlock situation; these events are succinctly described as follows:

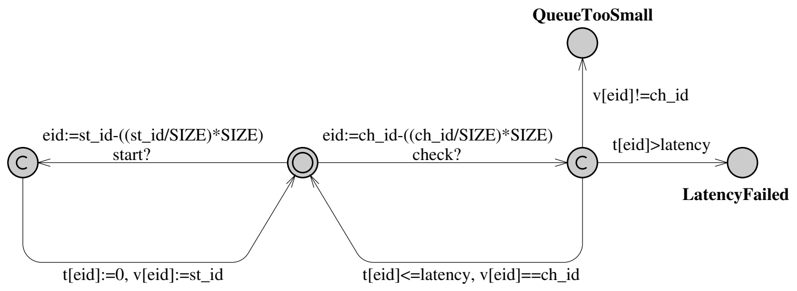


Fig. 6. An Automaton Template for Latency Obligation

1. The first video frame arrives at the video window, i.e. at a time 50ms after being generated.
2. The *videoReady* and *videoPresent* signals both happen with no (significant) time delay, also at time 50.
3. The video window completes presentation with no delay and the *videoPresented* signal is output immediately, i.e. at time 50.
4. The second frame arrives late at the video window, i.e. at a time 59ms after being generated, i.e. 49ms after the first was received and 99ms after the first was generated.
5. The *videoReady* and *videoPresent* signals occur immediately, i.e. at time 99 (relative to first frame generation).
6. The video window takes anything longer than 1ms to present the frame and the *videoPresented* signal is output at a time > 100ms (relative to first frame generation). This is >50ms since the last *videoPresented* signal, and does not meet the obliged anchoredJitter QoS constraint on the video window.
7. Analysis of this trace determines that the QoS specified for the video window constrains it to take *at most* 5ms to present a frame. Thus if this varies between 0ms and 5ms, on top of the expected 10ms jitter as input, it follows that the output will vary by more than 10ms. The situation could be aggravated further if the synchronisation algorithm delays the *videoPresent* signal by (the possible) 5ms within this scenario.

After analysing the scenario, the designer can determine a solution to the problem and alter the system design in order to remedy the problem. For instance, possible options to solve the problem detected in our example system are as follows:

1. Allow more jitter on the output of the video window,
2. Fix the time taken to present a frame – may not be possible if it is a hardware constraint,
3. Add a buffer on the video stream to reduce the input jitter.

For example, we could implement options 2 and 3. To do so we could a fix time for video frames to be presented, i.e. specify a fixed latency between signals *videoPresent* and *videoPresented*; and state that the anchored jitter on the output rate must be less than 15ms rather than 10 ms. These changes ensure that the jitter on frames being presented will be the same as the jitter on the signals instructing frames to be presented (coming out for the controller) and that the controller is free to add up to an extra 5ms of jitter on top of the possible 10ms caused by transmission over the binding. The changes to the specification are localised to the QoS profile for the video window, which is altered to that shown below. Using this, the resulting system is now free from this deadlock; absence of all such deadlocks indicates that the functional behaviour is consistent with the QoS specification.

```
QosProfile for VideoWindow {
  exp: quality {
   throughput(1000,recVideo.video) >= 25; };
  obl: quality {
   throughput(1000,vidReady.videoPresented)>=25;
   latency(recVideo.video,vidReady.videoReady)=0;
   latency(recVideo.video,vidReady.videoPresented)<=10;
   latency(vidCtrl.videoPresent,
                          vidReady.videoPresented)=5;
   anchoredJitter(vidReady.videoPresented) < 15; }; }
```

4.2 Feedback of Results

A weakness of model checking TA in the context of higher level languages, such as the UML, is that the output of model checking tools is a trace in the formal language of the model checker. For example, the trace output from the above example defines a sequence of events and transitions on TA with the time that they occur or were taken. A trace in this form is of little use to a designer working in the UML/OCL context who is not familiar with the level of formality provided by TA; the designer did not specify the system in TA. To overcome this we must be able to give feedback to the designer using the language in which he has designed the system.

Table 1 shows a trace of transitions (derived from UPPAAL) and the time at which the transitions are taken, for the situation leading to deadlock described in the previous sub-section. Synchronised transitions are shown as a pair in the same row of the table. A two-part label identifies the transitions. The first part is the instance name of the TA template in which the transition is defined; the second part, in UPPAAL is a number identifying a particular transition, we have replaced this with the name of either ‘internal’ if there is no synchronisation, or the parameter name (specified in the template) for the channel on which the synchronisation is defined.

The ODP computational viewpoint semantics defines inter-object communication to be in the form of signals. Our translation approach maps communication signals between objects to synchronisation events in a TA model and primitive bindings between interfaces map to the synchronisation channels in UPPAAL; thus the trace of events given by UPPAAL can be transformed back into a trace of inter-object communications (across interfaces) within the domain of the original (designers) language of the computational viewpoint. The original snapshot diagram that defines the configuration of system components can be used as a vehicle to illustrate back to the designer the problematic trace.

Table 1 shows some of the transitions highlighted by a darker background. These are the synchronised transitions that map to communication between objects; the other transitions are internal to an object (as indicated) or relate to internal communication between the audio and video parts of the synchronisation controller. These shaded transitions can be illustrated by a series of computational viewpoint snapshots, which we show in Fig. 7. This transformation appears to be automatable, and ongoing work is investigating implementation of an automatic transformer.

Table 1. UPPAAL Trace of transitions leading to deadlock

Time	Transition(s)
50	(videoSource.out, vidWin.videoIn)
50	(audioSource.out, speaker.audioIn)
50	(vidWin.videoReady, split.in)
50	(speaker.audioReady, synchController_Audio.audioReady)
50	(synchController.*internal*)
50	(split.out1, synchController_Video.vidReady)
50	(synchController_Video.*internal*)
50	(synchController_Video.*internal*)
50	(synchController_Video.*internal*)
50	(synchController_Video.vidPresent, vidWin.videoPresent)
50	(synchController_Video.*internal*)
50	(split.out2, synchController_Audio.vidReady)
50	(vidWin.videoPresented, vidOutput.in)
50	(synchController_Video.*internal*)
51	(synchController_Video.*internal*)
...	"
78	(synchController_Video.*internal*)
79	(vidOutput.*internal*)
79	(vidSource.*internal*)
79	(synchController_Video.*internal*)
80	(synchController_Video.*internal*)
81	(synchController_Video.*internal*)
82	(synchController_Video.*internal*)
83	(synchController_Video.*internal*)
84	(synchController_Video.*internal*)
84	(synchController_Video.*internal*)
99	(videoSource.out, vidWin.videoIn)
99	(vidWin.videoReady, split.in)
99	(split.out1, synchController_Video.vidReady)
99	(synchController_Video.*internal*)
99	(synchController_Video.*internal*)
99	(synchController_Video.*internal*)
99	(synchController_Video.vidPresent, vidWin.videoPresent)
99	(synchController_Video.*internal*)
99	(split.out2, synchController_Audio.vidReady)
100	(synchController_Video.*internal*)
101	(synchController_Video.*internal*)
102	(synchController_Video.*internal*)
103	(synchController_Video.*internal*)
104	(vidWin.videoPresented, vidOutput.in)
deadlock	***in state* vidOutput.RateTooSlow**

5 Conclusion

Our verification technique builds on previous works that propose the use of Timed Automata for modelling QoS and Statechart based behaviour specifications. [12]

maps UML state machines into UPPAAL as does [9]; either of these techniques would compliment our work, and give us mechanisms to transform the state diagram based functional behaviour into UPPAAL timed automata. A group at Lancaster have made significant use of timed automata for modelling distributed and multimedia systems: [5] discusses the use of temporal logic and timed automata for modelling such systems in a multi-paradigm approach and in [11] they specify systems directly in TA and map these to Java-beans to provide prototype implementations of the specified systems. The work of [7] shows ways to model QoS using timed automata and we have made use of that work to compliment our own.

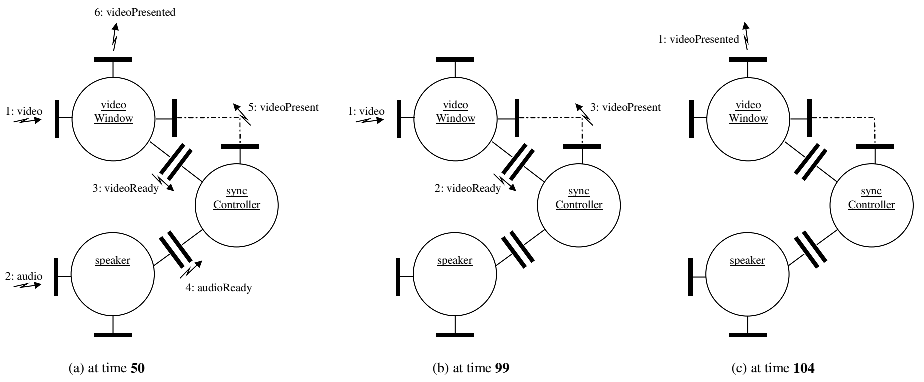


Fig. 7. Series of snapshots leading to deadlock / invalid QoS

There are two aspects to the work presented in this paper, the approach taken for designing distributed systems and the technique for verifying the QoS of the system.

Structurally, the proposed design languages enable the definition of all structural components from the computational viewpoint; to form a complete system specification it is necessary to provide specifications from the other four ODP viewpoints; this is left as future work.

Of course there are some limitations to the approach. For example, the specification of QoS is limited by the nature of the language chosen to express it, CQML. The primary limitation we discovered with this language, is that in the context of a computational object we use it to define constraints on interactions at each interface supported by an object; if an object supports an interface signature multiple times, there is no means in CQML to quantify over the collection of interfaces; we believe that CQML could be extended to enable such quantifications.

Future plans include provision of a design tool that incorporates our design languages and facilities to perform the verification and feed back of results. Additionally we are looking at techniques for inferring QoS constraints across interface bindings and through an objects functional behaviour. It is expected that this will enable us to analyse individual components of a particular configuration, separately - deducing similar results as if we had analysed the complete configuration. This should aid us in avoiding the huge state explosion problems we would get by attempting to model check large timed automata networks.

References

1. Aagedal J. Ø. "Quality of Service Support in Development of Distributed Systems," PhD thesis, Department of Informatics, Faculty of Mathematics and Natural Sciences, The University of Oslo, 2001
2. Akehurst D. H., Bordbar B., Derrick J., and Waters A. G., "Design and Verification of Distributed Multi-media Systems," University of Kent at Canterbury, 1-03, January 2003.
3. Akehurst D. H., Derrick J., and Waters A. G., "Addressing Computational Viewpoint Design," in proceedings EDOC 2003, Brisbane, Australia, September 2003.
4. Blair G. and Stefani J.-B., *Open Distributed Processing and Multimedia*: Addison Wesley, ISBN 0-201-17794-3, 1997.
5. Blair L., "The Role of Temporal Logic and Time Automata in Distributed Multimedia Systems," in proceedings Modal & Temporal Logic Based Planning for Open Networked Multimedia Systems (PONMS '99), Cape Cod, MA, pp. 1-7, November 1999.
6. Bordbar B., Derrick J., and Waters A. G., "Using UML to specify QoS constraints in ODP," *Computer Networks*, vol. 40, pp. 279-304, 2002.
7. Bowman H., Faconti G., Katoen J.-P., Latella D., and Massink M., "Automatic verification of a lip-synchronisation algorithm using uppaal - extended version," in B. Luttick, J. F. Groote, and J. V. Wamel (eds) proceedings FMICS'98 Third International Workshop on Formal Methods for Industrial Critical Systems, CWI, Amsterdam, The Netherlands, pp. 97-124, May 1998.
8. David A. and Moller M. O., "From HUppaal to Uppaal: A translation from hierarchical timed automata to flat timed automata," BRICS, Department of Computer Science, University of Aarhus,, Research Series RS-01-11, March 2001.
9. David A., Moller M. O., and Yi W., "Formal Verification of UML Statecharts with Real-Time Extensions," in R.-D. Kutsche and H. Weber (eds) proceedings 5th International Conference, FASE 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Springer, LNCS, Volume 2306, Grenoble, France, April 2002.
10. Harel D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, pp. 231-274, 1987.
11. Jones T. and Blair L., "Prototyping of Real-time Component Based Systems by the use of Timed Automata," in P. Hofmann and A. Schürr (eds) proceedings Workshop on Object-oriented Modeling of Embedded Real-time Systems (OMER-2 '01), GI-Edition - Lecture Notes in Informatics (LNI), P-5, Munich, Germany, May 2001.
12. Knapp A., Merz S., and Rauh C., "Model Checking Timed UML State MAchines and Collaborations," in W. Damm and E.-R. Olderog (eds) proceedings 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems, FTRTFT 2002, Co-sponsored by IFIP WG 2.2,, LNCS, Volume. 2469, Oldenburg, Germany, September 2002.
13. Larsen K. G., Pettersson P., and Yi W., "UPPAAL in a Nutshell," *Springer International Journal of Software Tools for Technology Transfer*, vol. 1, October 1997.
14. Waters A. G., Linington P. F., Akehurst D. H., Utton P., and Martin G., "Permabase: Predicting the performance of distributed systems at the design stage," *IEE Proceedings - Software*, vol. 148, pp. 113-121, August 2001.
15. X.641, "Information technology - Quality of service: Framework," vol. 1997: ITU-T Recommendation, 1998.
16. X.901-5, "Information Technology - Open Distributed Processing - Reference Model: All Parts," ITU-T Recommendation, 1996-99.

Author Index