Lecture Notes in Computer Science 3657

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

Editorial Board

David Hutchison
Lancaster University, UK

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Moni Naor
Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
University of Dortmund, Germany

Madhu Sudan
Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos
New York University, NY, USA

Doug Tygar
University of California, Berkeley, CA, USA

Moshe Y. Vardi
Rice University, Houston, TX, USA

Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Frank S. de Boer Marcello M. Bonsangue
Susanne Graf Willem-Paul de Roever (Eds.)

Formal Methods for Components and Objects

Third International Symposium, FMCO 2004
Leiden, The Netherlands, November 2 – 5, 2004
Revised Lectures

Volume Editors

Frank S. de Boer
CWI, Centre for Mathematics and Computer Science
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
E-mail: F.S.de.Boer@cwi.nl

Marcello M. Bonsangue
Leiden University
Leiden Institute of Advanced Computer Science
P.O. Box 9512, 2300 RA Leiden, The Netherlands
E-mail: marcello@liacs.nl

Susanne Graf
VERIMAG
Centre Equitation
2 Avenue de Vignate, 38610 Grenoble-Gières, France
E-mail: Susanne.Graf@imag.fr

Willem-Paul de Roever
Christian-Albrechts-University of Kiel
Institute of Computer Science and Applied Mathematics
Hermann-Rodewald-Straße 3, 24118 Kiel, Germany
E-mail: wpr@informatik.uni-kiel.de

Library of Congress Control Number: 2005932547

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

ISSN 0302-9743
ISBN-10 3-540-29131-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-29131-2 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springeronline.com

© Springer-Verlag Berlin Heidelberg 2005
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11561163 06/3142 5 4 3 2 1 0

Preface

Large and complex software systems provide the necessary infrastructure in all industries today. In order to construct such large systems in a systematic manner, the focus in the development methodologies has switched in the last two decades from functional issues to structural issues: both data and functions are encapsulated into software units which are integrated into large systems by means of various techniques supporting reusability and modifiability. This encapsulation principle is essential to both the object-oriented and the more recent component-based software engineering paradigms.

Formal methods have been applied successfully to the verification of medium-sized programs in protocol and hardware design. However, their application to the development of large systems requires more emphasis on specification, modelling and validation techniques supporting the concepts of reusability and modifiability, and their implementation in new extensions of existing programming languages.

In order to stimulate interaction between the different areas of software engineering and formal methods, with a special focus on component-based and object-oriented software systems, we organized the *3rd International Symposium on Formal Methods for Components and Objects (FMCO)* in Leiden, The Netherlands, from November 2nd to 5th, 2004. The program consisted of tutorial and technical presentations given by leading experts in the fields of theoretical computer science and software engineering. The symposium was attended by more than 75 people from all over the world.

This volume contains the contributions after the symposium of the invited speakers. We believe that the presented material provides a unique combination of ideas on software engineering and formal methods which reflect the expanding body of knowledge on modern software systems.

July 2005

F.S. de Boer
M.M. Bonsangue
S. Graf
W.-P. de Roever

Organization

The series of FMCO symposia are organized in the context of the bilateral NWO/DFG project Mobi-J and of the European IST project Omega.

The Mobi-J Project

Mobi-J is a project founded by a bilateral research program of The Dutch Organization for Scientific Research (NWO) and the Central Public Funding Organization for Academic Research in Germany (DFG). The partners of the Mobi-J projects are: the Centrum voor Wiskunde en Informatica, the Leiden Institute of Advanced Computer Science, and the Christian-Albrechts-Universität Kiel.

This project aims at the development of a programming environment which supports component-based design and verification of Java programs annotated with assertions. The overall approach is based on an extension of the Java language with a notion of component that provides for the encapsulation of its internal processing of data and composition in a network by means of mobile asynchronous channels.

The Omega Project

The overall aim of the European IST project Omega (2001-33522) is the definition of a development methodology in UML for embedded and real-time systems based on formal verification techniques. The approach is based on a formal semantics of a suitable subset of UML, adapted and extended where needed with a special emphasis on time-related aspects.

The Omega project involves the following partners: VERIMAG (France, Coordinator), Centrum voor Wiskunde en Informatica (The Netherlands), Christian-Albrechts-Universität (Germany), University of Nijmegen (The Netherlands), Weizmann Institute (Israel), OFFIS (Germany), EADS Launch Vehicles (France), France Télécom R&D (France), Israeli Aircraft Industries (Israel), and National Aerospace Laboratory (The Netherlands).

Sponsoring Institutions

The Dutch Organization for Scientific Research (NWO)
The European project IST-2001-33522 Omega
The Lorentz Center, Leiden, The Netherlands
The Royal Netherlands Academy of Arts and Sciences (KNAW)
The Dutch Institute for Programming Research and Algorithmics (IPA)
The Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands
The Leiden Institute of Advanced Computer Science (LIACS), The Netherlands

Table of Contents

A Theory of Predicate-Complete Test Coverage and Generation
Thomas Ball 1

A Perspective on Component Refinement
Luís S. Barbosa 23

A Fully Abstract Semanti cs for UML Components
Frank S. de Boer, Marcello M. Bonsangue, Martin Steffen, Erika Ábrahám 49

From (Meta) Objects to Aspects: A Java and AspectJ Point of View
Pierre Cointe, Hervé Albin-Amiot, Simon Denier 70

MoMo: A Modal Logic for Reasoning About Mobility
Rocco De Nicola, Michele Loreti 95

Probabilistic Linda-Based Coordination Languages
Alessandra Di Pierro, Chris Hankin, Herbert Wiklicky 120

Games with Secure Equilibria
Krishnendu Chatterjee, Thomas A. Henzinger, Marcin Jurdziński 141

Priced Timed Automata: Algorithms and Applications
Gerd Behrmann, Kim G. Larsen, Jacob I. Rasmussen 162

rCOS: Refinement of Component and Object Systems
Zhiming Liu, He Jifeng, Xiaoshan Li 183

Program Generation and Components
Davide Ancona, Eugenio Moggi 222

Assertion-Based Encapsulation, Object Invariants and Simulations
David A. Naumann 251

A Dynamic Binding Strategy for Multiple Inheritance and Asynchronously Communicating Objects
Einar Broch Johnsen, Olaf Owe 274

Observability, Connectivity, and Replay in a Sequential Calculus of Classes
Erika Ábrahám, Marcello M. Bonsangue, Frank S. de Boer, Andreas Grüner, Martin Steffen 296

Timing Analysis and Timing Predictability
Reinhard Wilhelm 317

Author Index 325

A Theory of Predicate-Complete Test Coverage and Generation

Thomas Ball

Microsoft Research, Redmond, WA, USA
tball@microsoft.com

Abstract. Consider a program with m statements and n predicates, where the predicates are derived from the conditional statements and assertions in a program. An *observable state* is an evaluation of the n predicates under some state at a program statement. The goal of *predicate-complete* testing (PCT) is to evaluate all the predicates at every program state. That is, we wish to cover every reachable observable state (at most $m \times 2^n$ of them) in a program. PCT coverage subsumes many existing control-flow coverage criteria and is incomparable to path coverage. To support the generation of tests to achieve high PCT coverage, we show how to define an upper bound U and lower bound L to the (unknown) set of reachable observable states R. These bounds are constructed automatically using Boolean (predicate) abstraction over modal transition systems and can be used to guide test generation via symbolic execution. We define a static coverage metric as $|L|/|U|$, which measures the ability of the Boolean abstraction to achieve high PCT coverage.

1 Introduction

Control-flow-based test generation generally has as its goal to cover all the statements or branches in a program. There are various control-flow adequacy criteria that go beyond branch coverage, such as multiple condition coverage, the ultimate of which is path coverage. Errors that go undetected in the face of 100% statement or branch coverage may be due to complex correlations between the predicates (that control the execution of statements) and the statements (that affect the value of these predicates) of a program. However, paths are notoriously difficult to work with as a coverage metric because there are an unbounded number of them in programs with loops, which characterizes most interesting programs in existence.

So, we seek an alternative to path coverage that has its "exhaustive" quality but induces a finite (rather than infinite) state space. We start with a fixed notation for atomic predicates (not containing Boolean connectives), taken from the relevant programming language. A predicate maps a state to a Boolean value. For example, the predicate $(x > 0)$ observes whether or not variable x has a positive value in a given state. Consider a program with m statements and n predicates. These predicates can be drawn from the conditional statements and assertions in a program, as well as from implicit run-time safety checks (for

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 1–22, 2005.
© Springer-Verlag Berlin Heidelberg 2005

checking for array bounds violations or divide-by-zero errors, for example). An *observable state* is an evaluation of the n predicates under some program state at a statement. While the set of states in a program is unbounded, the size of the set of observable states (S) is at most $(m \times 2^n)$.

The goal of *predicate-complete* testing (PCT) is to evaluate all the predicates at every program state. That is, we wish to cover all *reachable* observable states. PCT coverage is motivated by the observation that certain errors in a program only can be exposed by considering the complex dependences between the predicates in a program and the statements whose execution they control. The n predicates represent all the case-splits on the input that the programmer has identified (Of course, the programmer may have missed certain cases– specification-based testing would need to be used to determine the absence of such case-splits). In the limit, each of the m statements may have different behavior in each of the 2^n possible observable states, and so should be tested in each of these states. We show that PCT coverage subsumes traditional coverage metrics such as statement, branch and multiple condition coverage and that PCT coverage is incomparable to path coverage. PCT groups paths ending at a statement s into equivalence classes based on the observable states the paths induce at s.

Control-flow coverage metrics result from dividing a dynamic measure (for example, the number of statements executed by a test) into a static measure (for example, the number of statements in a program). Clearly, such a metric also can be defined for observable states. However, the choice of $(m \times 2^n)$ as a denominator will not do, as we expect many of the $(m \times 2^n)$ states to be unreachable. (Statement coverage does not suffer greatly from this problem because most statements are reachable). For example, if the set of predicates contains $(x = 0)$ and $(x = 1)$ then not all combinations are possible.

Thus, we desire a way to define a better denominator for PCT coverage. The main result of this paper is a way to overapproximate and underapproximate the set of reachable observable states (R) using the theory of modal transition systems and Boolean abstraction. The Boolean abstraction of a program with respect to its n predicates is a non-deterministic program, whereas the original concrete program is deterministic. We show how reachability analysis of this abstract program yields an upper bound U for R ($R \subseteq U$) as well as a lower bound L for R ($L \subseteq R$). The set U is an *overapproximation* of R: any state outside U is not a reachable observable state and need not (indeed, cannot) be tested. This set U provides a better denominator than $(m \times 2^n)$.

Conversely, the set L is an *underapproximation* of R: any state in L *must* be a reachable observable state. That is, every state in L must be testable. We show how to use L to guide symbolic path-based test generation to cover the untested states in L. Our definition of L relies on a novel enhancement to modal transition systems that enlarges the set of states that can be proved to be reachable observable states.

This paper is organized as follows. Section 2 compares predicate-complete test coverage to other forms of control-flow coverage. Section 3 precisely defines

the system of abstraction we will use to compute the upper and lower bounds. Section 4 gives algorithms to compute these upper and lower bounds. Section 5 gives an example that shows how the use of PCT coverage and the lower and upper bounds can be used to expose an error in a small function. Section 6 presents an algorithm that uses the lower bound to guide test generation. Section 7 discusses some of the implications of our results. Section 8 reviews related work and Section 9 concludes the paper.

2 A Characterization of Predicate-Complete Test Coverage

This section compares PCT coverage with other forms of control-flow coverage. In this comparison, we decompose complex predicates into atomic predicates. So, the program fragment "`L1: if ((x<0)||(y<0)) S else T`" contains two branches corresponding to the atomic predicates `(x<0)` and `(y<0)`. Based on this decomposition, the concepts of branches, atomic predicates and conditions are equivalent.

To recap, complete PCT coverage means that each reachable observable state of a program is covered by a test. This implies that each (executable) statement is executed at least once, so PCT subsumes statement coverage. PCT coverage requires that each predicate be tested so as to evaluate to both true and false (of course this may not be possible for unsatisfiable predicates such as `(x!=x)`), so it subsumes branch coverage. PCT clearly also subsumes multiple condition coverage and its variants. Considering the program fragment given above, multiple condition coverage requires every possible Boolean combination of `(x<0)` and `(y<0)` to be tested at `L1`, which seems similar to PCT. But now, consider the sequencing of two **if** statements:

```
L2: if (A || B) S else T
L3: if (C || D) U else V
```

PCT requires that every Boolean combination over the set { `A`, `B`, `C`, `D` } be tested at every statement in the program (six in this case, the two **if** statements and the four statements `S`, `T`, `U` and `V`). Multiple condition coverage only requires that every Boolean combination over { `A`, `B` } be tested at `L2` and that every that every Boolean combination over { `C`, `D` } be tested at `L3`. Similarly, predicate-based test generation [Tai96, Tai97] focuses on testing predicates in a program. It considers correlations between predicates that appear in a single conditional statement but does not consider correlations between predicates that appear in different conditional statements, as does PCT.

Of course, we can view paths as possible logical combinations of predicates, so it is natural to ask how PCT relates to path coverage. As a loop-free program with n predicates can have at most 2^n paths, it seems like PCT might have the ability to explore more behaviors (as it may explore $m \times 2^n$ states in the limit). In fact, we show PCT and path coverage are incomparable, even for loop-free programs.

```
                  // reachable observable states
L0:  y = 0;       // (L0,x<0)     (L0,!(x<0))
L1:  if (x<0)     // (L1,x<0)     (L1,!(x<0))
L2:    skip;      // (L2,x<0)
     else         //
L3:    x = -2;    //              (L3,!(x<0))
L4:  x = x + 1;   // (L4,x<0)
L5:  if (x<0)     // (L5,x<0)     (L5,!(x<0))
L6:    y = 1;     // (L6,x<0)
```

(a)

```
L1:  if (p)
L2:    if (q)
L3:      x=0;
L4:  B;
```

(b)

Fig. 1. (a) A program that shows it is possible to attain full PCT coverage without covering all feasible paths. (b) A program that that shows it is possible to cover all feasible paths without attaining full PCT coverage.

The program in Figure 1(a) shows that it is possible to cover all reachable observable states in a (loop-free) program without covering all feasible paths. In this program, we assume that the uninitialized variable `x` can take on any initial (integer) value. The program has one predicate `(x<0)`. Thus an observable state of this program is a pair of a program counter (program label) and a value for the predicate `(x<0)`.

The reachable observable states of this program are shown in the comments to the right of the program. The set of tests $\{ x \rightarrow -1, x \rightarrow 1 \}$ covers all these states. The test $\{ x \rightarrow -1 \}$ covers the observable states

$$\{ \text{(L0,x<0)}, \text{(L1,x<0)}, \text{(L2,x<0)}, \text{(L4,x<0)}, \text{(L5,!(x<0))} \}$$

via the path (L0,L1,L2,L4,L5), while the test $\{ x \rightarrow 1 \}$ covers the observable states

$$\{ \text{(L0,!(x<0))}, \text{(L1,!(x<0))}, \text{(L3,!(x<0))}, \text{(L4,x<0)}, \text{(L5,x<0)}, \text{(L6,x<0)} \}$$

via the path (L0,L1,L3,L4,L5,L6). However, this set of tests does not cover the feasible path (L0,L1,L2,L4,L5,L6), which is covered by the test $\{ x \rightarrow -2 \}$.

Because of the assignment statement "`x = -2;`", the set of reachable observable states at label L4 (namely (L4,x<0)) cannot distinguish whether the executed path to L4 traversed the **then** or **else** branch of the initial **if** statement. While PCT can track many correlations, assignment statements such as the one above can cause PCT to lose track of correlations captured by path coverage.

In this example, if we add the predicate `(x==-2)` to the set of observed predicates then PCT coverage is equivalent to path coverage, as PCT coverage will require the test $\{ x \rightarrow -2 \}$ in order to cover the reachable state (L2,x==-2). It is an open question whether we can always find a minimal set of predicates for

which PCT coverage implies path coverage (or decide that only infinitely many predicates will do).[1]

Figure 1(b) shows that it is possible to cover all feasible paths in a (loop-free) program without covering all reachable observable states. The program has three feasible paths: (`L1`,`L2`,`L3`,`L4`), (`L1`,`L2`,`L4`) and (`L1`,`L4`). However, a test set of size three that covers these paths clearly will miss either the observable state (`L4`,`!p&&q`) or (`L4`,`!p&&!q`).

In summary, PCT coverage is a new type of coverage criteria that subsumes statement, branch, multiple condition and predicate coverage. PCT has similarities to path coverage but is strictly incomparable, as the above examples demonstrate. Section 8 compares PCT coverage to several other control-flow coverage criteria.

3 Formalizing Abstraction

In this section, we define the concepts of concrete and abstract transition systems that we will use to compute the upper and lower bounds, U and L, to the set of reachable observable states R of a program.

3.1 Concrete Transition Systems

We represent a deterministic sequential program by a concrete transition system (CTS) as follows:

Definition 3.1: (*Concrete Transition System*). A concrete transition system is a triple $(S_C, I_C, \longrightarrow)$ where S_C and I_C are non-empty sets of states and $\longrightarrow \subseteq S_C \times S_C$ is a transition relation satisfying the following constraints:

- $S_C = \{halt,\ error\} \cup T_C$, where T_C is a non-empty set of states;
- $I_C \subseteq T_C$ is the set of initial states;
- $\forall s_c \in T_C,\ |\{s'_c \in S_C \mid s_c \longrightarrow s'_c\}| = 1$

There are two distinguished end states, *halt* and *error*, which correspond to execution terminating normally and going wrong, respectively. These two states have no successor states. All other states have exactly one successor. Thus, a CTS models a program as a set of traces.

3.2 Abstract Transition Systems

Modal Transition Systems (MTSs) [GR03] are a formalism for reasoning about partially defined systems that we will use to model (Boolean) abstractions of CTSs. We generalize modal transition systems to tri-modal transition systems (TTSs) as follows:

[1] Thanks to Orna Kupferman for suggesting this question.

Definition 3.2: (*Tri-Modal Transition System*). A TTS is a tuple $(S, \overset{may}{\longrightarrow}, \overset{must^+}{\longrightarrow}, \overset{must^-}{\longrightarrow})$ where S is a nonempty set of states and $\overset{may}{\longrightarrow} \subseteq S \times S$, $\overset{must^+}{\longrightarrow} \subseteq S \times S$ and $\overset{must^-}{\longrightarrow} \subseteq S \times S$ are transition relations such that $\overset{must^+}{\longrightarrow} \subseteq \overset{may}{\longrightarrow}$ and $\overset{must^-}{\longrightarrow} \subseteq \overset{may}{\longrightarrow}$.

A total-onto abstraction relation[2] ρ induces an abstract TTS M_A from a CTS M_C as follows [God03]:

Definition 3.3: (*Precise Abstraction Construction*). Let $M_C = (S_C, I_C, \longrightarrow)$ be a CTS. Let S_A be a set of abstract states and ρ be a total-onto abstraction relation over pairs of states in $S_C \times S_A$. A TTS $M_A = (S_A, \overset{may}{\longrightarrow}_A, \overset{must^+}{\longrightarrow}_A, \overset{must^-}{\longrightarrow}_A)$ is constructed from M_C, S_A and ρ as follows:

(a) $s_a \overset{may}{\longrightarrow}_A s'_a$ **iff** $\exists (s_c, s_a) \in \rho : \exists (s'_c, s'_a) \in \rho : s_c \longrightarrow s'_c$;
(b) $s_a \overset{must^+}{\longrightarrow}_A s'_a$ **iff** $\forall (s_c, s_a) \in \rho : \exists (s'_c, s'_a) \in \rho : s_c \longrightarrow s'_c$;
(c) $s_a \overset{must^-}{\longrightarrow}_A s'_a$ **iff** $\forall (s'_c, s'_a) \in \rho : \exists (s_c, s_a) \in \rho : s_c \longrightarrow s'_c$;

It is easy to see that the definition of M_A satisfies the constraints of a TTS, namely that $\overset{must^+}{\longrightarrow}_A \subseteq \overset{may}{\longrightarrow}_A$ and $\overset{must^-}{\longrightarrow}_A \subseteq \overset{may}{\longrightarrow}_A$.

We have emphasized the "**iff**" (if-and-only-if) text to make the point that we assume it is possible to create a most precise abstract TTS M_A from a given CTS M_C. In general, this assumption does not hold for infinite-state systems. It does hold for the examples we consider here.

Figure 2 illustrates the three types of transitions in a TTS M_A constructed from a CTS M_C via the above definition. In this figure, the grey nodes represent states in S_C and edges between the grey nodes represent transitions in $\longrightarrow$. The dotted circles to the right represent the abstract states in M_A that the concrete states map to under the abstraction relation ρ. Let us examine the four cases in Figure 2:

- Case (a) shows a transition $a \overset{may}{\longrightarrow}_A a'$. *May*-transitions are depicted as dashed edges. This transition exists because concrete state d maps to a (under ρ) and transitions to d' via $d \longrightarrow d'$, where d' maps to a'. Note that there is no transition $a \overset{must^+}{\longrightarrow}_A a'$ or $a \overset{must^-}{\longrightarrow}_A a'$.
- Case (b) shows a transition $a \overset{must^+}{\longrightarrow}_A a'$, depicted as a solid edge with a "+" label. This transition exists because for all states $x \in \{b, c, d\}$ (mapping to a under ρ), there is a y' such that transition $x \longrightarrow y'$ exists (namely, $y' = d'$). That is, $must^+$-transitions identify a *total* relation between sets of concrete states corresponding to a and a'. Note that there is no transition $a \overset{must^-}{\longrightarrow}_A a'$.
- Case (c) shows a transition $a \overset{must^-}{\longrightarrow}_A a'$, which exists because for all states $y' \in \{c', d'\}$ (mapping to a' under ρ), there is an x such that $x \longrightarrow y'$ exists

[2] A total-onto relation over $D \times E$ contains at least one pair (d, e), $e \in E$, for each element $d \in D$ (it is total) and at least one pair (d', e'), $d' \in D$, for each element $e' \in E'$ (it is onto).

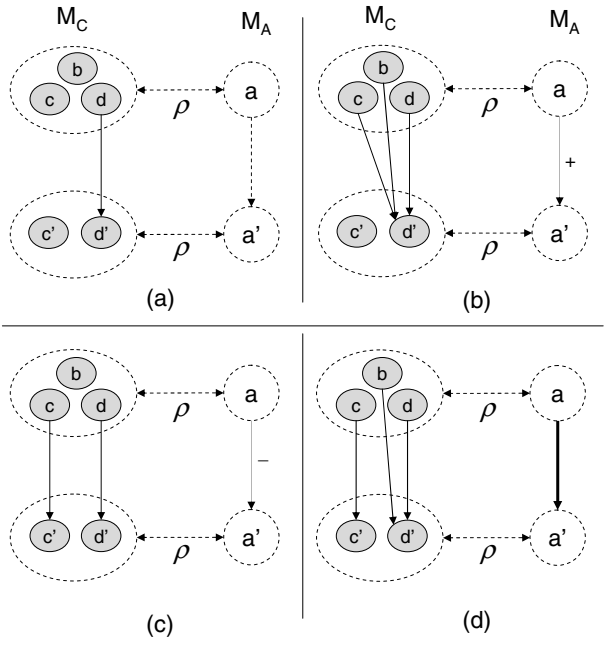


Fig. 2. Illustrations of (a) a *may*-transition; (b) a $must^+$-transition; (c) a $must^-$-transition; (d) a transition that is a $must^+$-transition and a $must^-$-transition.

(namely $x = c$ for $y' = c'$ and $x = d$ for $y' = d'$). That is, $must^-$-transitions identify an *onto* relation between sets of concrete states corresponding to a and a'. These transitions are depicted as solid edges with "-" labels.

– Case (d) shows the case in which there are both transitions $a \xrightarrow{must^+}_A a'$ and $a \xrightarrow{must^-}_A a'$. Let $a \xrightarrow{must^\#}_A a'$ denote the fact that $a \xrightarrow{must^+}_A a'$ and $a \xrightarrow{must^-}_A a'$. These transitions are depicted as bold edges.

3.3 Predicate Abstraction

Predicate abstraction maps a (potentially infinite-state) CTS into a finite-state TTS via a finite set of quantifier-free formulas of first-order logic $\Phi = \{\phi_1, \cdots, \phi_n\}$. A bit vector b of length n ($b = b_1 \cdots b_n$, $b_i \in \{0, 1\}$) defines an abstract state whose corresponding concrete states are those satisfying the conjunction $\langle b, \Phi \rangle = (l_1 \wedge \cdots \wedge l_n)$ where $l_i = \phi_i$ if $b_i = 1$ and $l_i = \neg\phi_i$ if $b_i = 0$. We write $s \models \langle b, \Phi \rangle$ to denote that $\langle b, \Phi \rangle$ holds in state s.

Definition 3.4: (*Predicate Abstraction of a CTS*). Given a CTS $M_C = (S_C, I_C, \longrightarrow)$ and a set of predicates $\Phi = \{\phi_1, \cdots, \phi_n\}$, predicate abstraction defines the total-onto abstraction relation ρ and the set of abstract states S_A:

– $\rho \in (S_C, \{0, 1\}^n)$, where $(s, b) \in \rho \iff s \models \langle b, \Phi \rangle$
– $S_A = \{b \in \{0, 1\}^n \mid \exists (s, b) \in \rho\}$

which define the finite-state abstract TTS $M_A = (S_A, \xrightarrow{may}_A, \xrightarrow{must^+}_A, \xrightarrow{must^-}_A)$ (per Definition 3.3). We assume that S_A contains abstract states $halt_A$ and $error_A$ that are in a one-to-one relationship with their counterparts *halt* and *error* from S_C.

It is useful to define an abstraction and concretization functions relating states in S_A to states in S_C:

Definition 3.5: (*Abstraction/Concretization Function*). Let $\rho : S_C \times S_A$ be an abstraction relation. Let $\alpha_\rho(C) = \{s_a \mid \exists s_c \in C : (s_c, s_a) \in \rho\}$ be the abstraction function mapping a set of concrete states to its corresponding set of abstract states. Let $\gamma_\rho(A) = \{s_c \mid \exists s_a \in A : (s_c, s_a) \in \rho\}$ be the concretization function mapping a set of abstract states to its corresponding set of concrete states. When ρ is understood from context we will use α in place of α_ρ and γ in place of γ_ρ. The set of initial abstract states I_A of M_A can then be defined as $I_A = \alpha(I_C)$.

3.4 Predicate Abstraction of Programs

Algorithms for computing the *may*- and $must^+$-transitions of a predicate abstraction of an MTS are given by Godefroid, Huth and Jagadeesan [GHJ01]. Computation of the $must^-$-transitions can be done in a similar fashion. Computation of the most precise abstract transitions is undecidable, in general. As usual, we assume the existence of a complete theorem prover that permits the computation of the most precise abstract transitions.

We review the basic idea here, where M_C is a program where a concrete state $c \in S_C$ gives a valuation to a program counter *pc* (ranging over a finite set of program locations) and a valuation to each program variable.

Let $WP(s, e)$ be the weakest pre-condition of a statement s with respect to expression e and let $SP(s, e)$ be the strongest post-condition of s with respect to e [Gri81]. (For any state c_1 satisfying $WP(s, e)$ the execution of s from c_1 results in a state c_2 satisfying e. For any state c_1 satisfying e the execution of s from c_1 results in a state c_2 satisfying $SP(s, e)$).

Let P_1 and P_2 be the concretization of two bit vectors b_1 and b_2 (i.e., $P_1 = \langle b_1, \Phi \rangle$ and $P_2 = \langle b_2, \Phi \rangle$). Let pc_1 and pc_2 be two program counters such that statement s is executed at pc_1 and pc_2 is a possible control successor of statement s. Statement s induces a *may*-transition from (pc_1, b_1) to (pc_2, b_2) if $\exists V : P_1 \wedge WP(s, P_2)$, where V is the set of free variables in the quantified expression.[3] Statement s induces a $must^+$-transition from (pc_1, b_1) to (pc_2, b_2) if $\forall V.P_1 \implies WP(s, P_2)$. Finally statement s induces a $must^-$-transition from (pc_1, b_1) to (pc_2, b_2) if $\forall V.P_2 \implies SP(s, P_1)$.

A naïve algorithm for computing all the abstract transitions of a program is to consider for each statement s all the possible pairs bit vectors ($b_1 \in 2^n$, $b_2 \in 2^n$) and use the method in the previous paragraph to determine the abstract

[3] Note that we have somewhat abused notation by including the program counter in the abstract state. However, since the program counter ranges over a finite set of n locations, we can encode the program counter with $log_2 n$ bits.

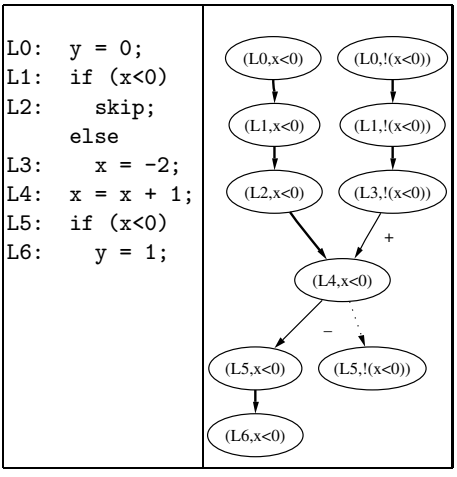


Fig. 3. (a) The program from Figure 1(a) and (b) its abstract transitions

transition(s) that s induces between P_1 and P_2 (the concretizations of b_1 and b_2). The cost of this algorithm is $O(m2^n)$.

3.5 Example

Figure 3(a) shows the program from Figure 1(a) and its set of (reachable) abstract transitions. Let us consider the statements in the program and the abstract transitions that they induce. The assignment statement at `L0` is "`y=0`". We have that $SP($`y=0,(x<0)`$) = WP($`y=0,(x<0)`$) =$ `(x<0)`. Therefore, we have a $must^{\#}$-transition `(L0,x<0)` $\stackrel{must^{\#}}{\longrightarrow}$ `(L1,x<0)`. For similar reasons, we have the $must^{\#}$-transition `(L0,!(x<0))` $\stackrel{must^{\#}}{\longrightarrow}$ `(L1,!(x<0))`.

The next statement is the `if`-statement at label `L1`. Because this statement branches exactly on the predicate `(x<0)`, it induces the $must^{\#}$-transitions `(L1,x<0)` $\stackrel{must^{\#}}{\longrightarrow}$ `(L2,x<0)` and `(L1,!(x<0))` $\stackrel{must^{\#}}{\longrightarrow}$ `(L3,!(x<0))`. The statement at label `L2` is a skip and so has no affect on the state, inducing the transition `(L2,x<0)` $\stackrel{must^{\#}}{\longrightarrow}$ `(L4,x<0)`.

The assignment statement at label `L3` is reachable only when `!(x<0)` is true. It assigns the value -2 to variable `x`. We have that $WP($`x=-2,(x<0)`$) =$ `(-2<0)`, which reduces to true. This means that there is a $must^{+}$-transition `(L3,!(x<0))` $\stackrel{must^{+}}{\longrightarrow}$ `(L4,(x<0)`. However, $WP($`x=-2,!(x<0)`$) =$ `(!(-2<0))`, which reduces to false. So there can be no transition from `(L3,!(x<0))` to (L4,`!(x<0)`). Now, let us consider strongest post-conditions. We have that $SP($`x=-2,!(x<0)`$) =$ `!(-2<0)`, which reduces to false, so there can be no $must^{-}$-transition from `(L3,!(x<0))` to `(L4,(x<0)`.

We now consider the assignment statement at label `L4` which is reachable only under `(x<0)` and which increments variable `x`. Because $SP($`x=x+1,(x<0)` $=$

`(x<1)` and the set of states satisfying `(x<0)` is a subset of the set of states satisfying `(x<1)`, there is a $must^-$-transition `(L4,x<0)` $\stackrel{must^-}{\longrightarrow}$ `(L5,x<0)`. There is no $must^+$-transition between these states because $WP($`x=x+1,(x<0)` $=$ `(x<-1)` and the set of states satisfying `(x<0)` is not a subset of the set of states satisfying `(x<-1)`. The assignment statement induces a *may*-transition `(L4,x<0)` $\stackrel{may}{\longrightarrow}$ `(L5,!(x<0))`, because this transition only takes places when variable `x` has the value -1 before the increment and the (resulting) value 0 after the increment.

Finally, there is a $must^\#$-transition `(L5,x<0)` $\stackrel{must^\#}{\longrightarrow}$ `(L6,x<0)` because the if-statement at label `L5` tests exactly the condition `(x<0)`.

4 Defining the Upper and Lower Bounds

Recall that the goal of predicate-complete testing (PCT) is to cover all reachable observable states, as defined by the m statements and n predicates $\Phi = \{\phi_1, \cdots, \phi_n\}$ in the program represented by the CTS M_C. The set of reachable observable states R is unknown, so we will use the Boolean (predicate) abstraction of M_C with respect to Φ to construct an abstract TTS M_A via the abstract relation ρ induced by Φ (see Definition 3.4).

We now show how to analyze M_A to compute both upper and lower bounds to R. To do so, we find it useful to define a reachability function for a transition system. Let S be a set of states and δ be a transition relation of type $S \times S$. We define the reachability function over δ and $S' \subseteq S$ as $reach[\delta](S') = \mu X.(S' \cup \delta(X))$, where μ is the least fixpoint operator and $\delta(X)$ is the image of set X under δ.

We now define reachability in a CTS: Let M_C be a CTS. We denote the set of states reachable from states in T ($T \subseteq S_C$) as $reach_C(T) = reach[\longrightarrow](T)$. That is, reachability in M_C is simply defined as the transitive closure over the transitions in M_C, starting from states in T. We then have that $R = \alpha(reach_C(I_C))$, where I_C is the set of initial states of M_C.

4.1 Upper Bound Computation

May-reachability in TTS M_A defines the upper bound U. Let M_C be a CTS and let M_A be an abstract TTS defined by abstraction relation ρ (via Definition 3.3). The upper bound is defined as $U = reach[\stackrel{may}{\longrightarrow}_A](I_A)$, where $I_A = \alpha(I_C)$. That is, U is simply defined as the transitive closure over the *may*-transitions in M_A from the initial states I_A. It is easy to see that $\alpha(reach_C(I_C)) \subseteq U$, as the *may*-transitions of M_A overapproximate the set of transitions in M_C (by Definition 3.3).

4.2 Lower Bound Computation *L*

A set of abstract states $X \subseteq S_A$ is a lower bound of R if for each $x_a \in X$, there is a $(x_c, x_a) \in \rho$ such that $x_c \in reach_C(I_C)$. This implies that $X \subseteq R$, as expected.

We define the lower bound L (based on analysis of M_A) to be:

$$L = \{\ v_a \ \mid\ \exists t_a, u_a : t_a \in reach[\overset{must^-}{\longrightarrow_A}](I_A) \quad \wedge \\ (t_a \overset{may}{\longrightarrow_A} u_a \ \vee \ t_a = u_a) \wedge \\ v_a \in reach[\overset{must^+}{\longrightarrow_A}](\{u_a\}) \quad \}$$

That is, an abstract state v_a is in L if there is a (possibly empty) sequence of $must^-$-transitions leading from $s_a \in I_A$ to t_a, there is a *may*-transition from t_a to u_a (or t_a is equal to u_a), and there is a (possibly empty) sequence of $must^+$-transitions from u_a to v_a.

We now show that for each $v_a \in L$, there is a $(v_c, v_a) \in \rho$ such that $v_c \in reach_C(I_C)$. That is, L is a lower bound to R. The proof is done in three steps, corresponding to the three parts of the definition of L:

- First, consider a sequence of $must^-$-transitions leading from $s_a \in I_A$ to t_a in M_A. Each $must^-$-transition $x_a \overset{must^-}{\longrightarrow_A} y_a$ identifies an *onto* relation from $\gamma(x_a)$ to $\gamma(y_a)$. That is, for all concrete states y_c mapping to y_a, there is a transition $x_c \longrightarrow y_c$ such that x_c maps to x_a. The transitive closure of an onto relation yields an onto relation. So, for all t_c mapping to t_a, we know that $t_c \in reach_C(I_C)$.
- Second, by the construction of M_A from M_C there is a *may*-transition $t_a \overset{may}{\longrightarrow_A} u_a$ only if there exists a transition $t_c \longrightarrow u_c$, where states t_c and u_c map to t_a and u_a, respectively. Since for all t_c mapping to t_a we know that $t_c \in reach_C(I_C)$, it follows that if there is a *may*-transition $t_a \overset{may}{\longrightarrow_A} u_a$ then there is some u_c mapping to u_a such that $u_c \in reach_C(I_C)$.
- Third, consider a sequence of $must^+$-transitions leading from t_a to v_a in M_A. Each $must^+$-transition $x_a \overset{must^+}{\longrightarrow_A} y_a$ identifies a *total* relation from $\gamma(x_a)$ to $\gamma(y_a)$. That is, for all concrete states x_c mapping to x_a, there is a transition $x_c \longrightarrow y_c$ such that y_c maps to y_a. The transitive closure of a total relation yields a total relation. So, for all t_c mapping to t_a, we know that there is a v_c mapping to v_a such that $v_c \in reach_C(\{t_c\})$. □

5 Example

This section demonstrates upper and lower bounds to the reachable observable states of a small function. Figure 4(a) presents a (buggy) example of QuickSort's `partition` function, a classic example that has been used to study test generation [BEL75]. We have added various control points and labels to the code for explanatory purposes. The goal of the function is to permute the elements of the input array so that the resulting array has two parts: the values in the first part are less than or equal to the chosen pivot value `a[0]`; the values in the second part are greater than the pivot value.

There is an array bound check missing in the code that can lead to an array bounds error: the check at the **while** loop at label `L2` should be `(lo<=hi &&`

```
void partition(int a[], int n) {
    assume(n>2);
    int pivot = a[0];
    int lo = 1;
    int hi = n-1;
L0: while (lo <= hi) {
L1:   ;
L2:   while (a[lo] <= pivot) {
L3:     lo++;
L4:     ;
      }
L5:   while (a[hi] > pivot) {
L6:     hi--;
L7:     ;
      }
L8:   if (lo < hi) {
L9:     swap(a,lo,hi);
LA:     ;
      }
LB:   ;
    }
LC: ;
}
```

Fig. 4. The `partition` function

`a[lo]<=pivot)`.[4] This error only can be uncovered via an input array in which all the elements of the array `a` have a value less than or equal to `a[0]`.

There are thirteen labels in the `partition` function (L0-LC), but an unbounded number of paths. Instead of reasoning in terms of paths, we will use predicates to observe the states of the `partition` function. Let us observe the four predicates that appear in the conditional guards of the function: `(lo<hi)`, `(lo<=hi)`, `(a[lo]<=pivot)` and `(a[hi]>pivot)`. An observed state thus is a bit vector of length four (**lt**, **le**, **al**, **ah**), where **lt** corresponds to `(lo<hi)`, **le** corresponds to `(lo<=hi)`, **al** corresponds to `(a[lo]<=pivot)`, and **ah** corresponds to `(a[hi]>pivot)`. There only are ten feasible valuations for this vector, as six are infeasible because of correlations between the predicates.[5] These correlations reduce the possible observable state space from $13 * 16 = 208$ to $13 * 10 = 130$.

[4] The loop at `L5` cannot decrement `hi` to take a value less than zero because the value of variable `pivot` is fixed to be the value of `a[0]`. One could argue that one would want to put a bounds check in anyway.

[5] Since `(lo<hi)` implies `(lo<=hi)`, the four valuations TFFF, TFTT, TFFT and TFTF are infeasible. Also, if `!(lo<hi)&&(lo<=hi)` then `(lo==hi)` and so exactly one of the predicates in the set { `(a[lo]<=pivot)`, `(a[hi]>pivot)` } must be true. Thus, the two valuations FTFF and FTTT are infeasible.

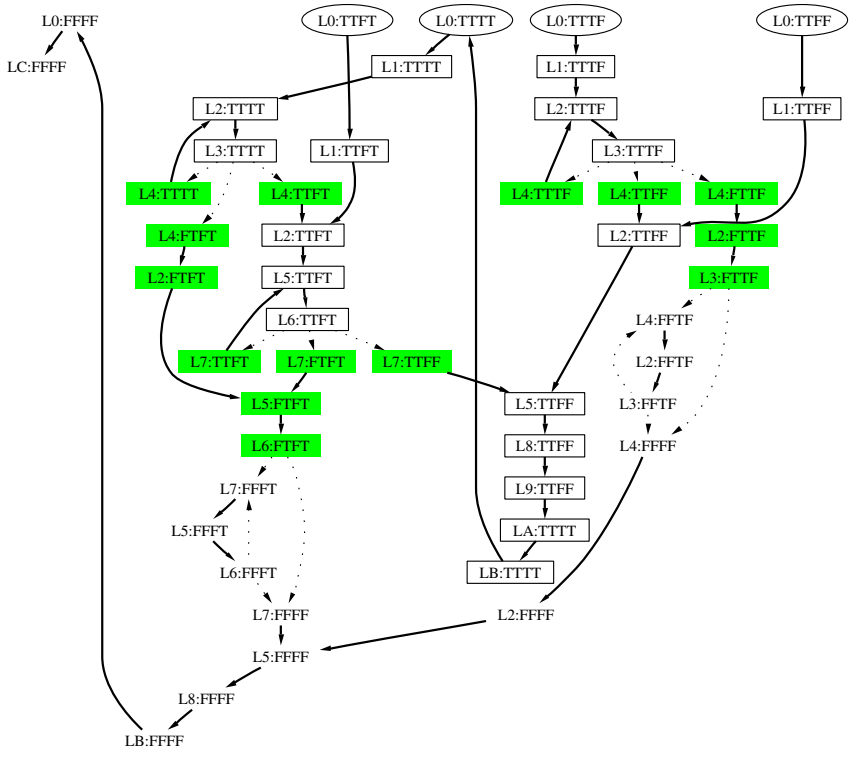


Fig. 5. The reachable abstract state space of `partition` function. The ovals represent the initial states I_A = { `L0`:TTFT, `L0`:TTTT, `L0`:TTTF, `L0`:TTFF }. The ovals and rectangles comprise the lower bound L, while the plaintext nodes represent the set $U - L$.

Figure 5 shows the upper and lower bounds of the program as a graph of abstract states. Each state is uniquely labeled `LX`:ABCD, where `LX` is the label (program counter), and A, B, C and D are the values of the Boolean variables `lt`, `le`, `al`, and `ah`. The (four) initial abstract states (I_A) are denoted by ovals. Consider the initial state `L0`:TTTT. This abstract state corresponds to all concrete states that satisfy the expression:

```
(lo<hi) && (a[lo]<=pivot) && (a[hi]>pivot)
```

Each edge in the graph represents a transition between two reachable abstract states (induced by the statement at the label of the first state). Solid edges represent transitions that are $must^{\#}$ (in this example, there are no transitions that are only $must^{+}$ or only $must^{-}$). Dotted edges represents $\overset{may}{\longrightarrow}$ transitions.

The set of nodes in Figure 5 represent the states that comprise the upper bound U ($|U| = 49$). The rectangular nodes represent the set L ($|L| = 35$) and the plaintext nodes represent the set $U - L$. The shading of the rectangular nodes indicates the following:

- The white rectangular nodes represent those abstract states reachable from I_A via a sequence of $must^-$-transitions (in our example, these are $must^\#$-transitions which are, by definition, $must^-$-transitions). For example, consider the initial state `L0`:TTTF. There is a path of $must^\#$-transitions `L0`:TTTF $\stackrel{must^\#}{\longrightarrow_A}$ `L1`:TTTF $\stackrel{must^\#}{\longrightarrow_A}$ `L2`:TTTF $\stackrel{must^\#}{\longrightarrow_A}$ `L3`:TTTF.
- The light-grey rectangular nodes (green in color) represent those abstract states only reachable via a sequence of $must^-$-transitions, followed by one *may*-transition, followed by a sequence of $must^+$-transitions. Thus, the set of ovals plus the set of white and light-grey rectangular nodes represents the set L. Consider the *may*-transition `L3`:TTTF $\stackrel{may}{\longrightarrow_A}$ `L4`:FTTF, which continues the path given above. Covering this transition is the only way in which the state `L4`:FTTF can be reached. Then there is a path of $must^\#$-transitions (which, by definition, also are $must^+$-transitions): `L4`:FTTF $\stackrel{must^\#}{\longrightarrow_A}$ `L2`:FTTF $\stackrel{must^\#}{\longrightarrow_A}$ `L3`:FTTF. So, these three nodes are colored light-grey.

The path given above is one of the paths that leads to an array bounds error. Note that in this path the label `L3` occurs twice, once in the state `L3`:TTTF and then in the state `L3`:FTTF. In the first state, we have that `(lo<=hi)`, `(a[lo]<=pivot)` and `(a[hi]<=pivot)`. At label `L3`, `lo` is incremented by one. The path dictates (via the *may*-transition `L3`:TTTF $\stackrel{may}{\longrightarrow_A}$ `L4`:FTTF) that the value of `lo` and `hi` are now equal. Because `(a[hi]<=pivot)` the loop at label `L2` continues to iterate and we reach the second state, `L3`:FTTF, in which we have that `(lo==hi)` and `(a[lo]<=pivot)` and `(a[hi]<=pivot)`. When `lo` is incremented the second time, its value becomes greater than `hi`, whose value still is the index of the last element of the array. Thus, the next access of `a[lo]` at label `L2` is guaranteed to cause an array bounds violation.

6 Test Generation

The goal of test generation is to cover all the states in the lower bound L (plus any additional states, if we are lucky). Our test generation process consists of three steps:

- *Path Generation*: we use the set L to guide test generation. In particular, using this set, we identify a set of paths that are guaranteed to cover all states in L;
- *Symbolic Execution*: we use symbolic execution on this set of paths in order to generate test data to cover these paths;
- *Observe Test Runs*: the program under test is run against this set of tests to check for errors and collect the set of executed observable states.

6.1 Path Generation

Let I_A be the set of initial abstract states in M_A. Consider the set of states L. The goal of the path generation phase is to enumerate all paths from I_A

Path Endpoints	Generated Input Array	Bounds Error?
(L0:TTTT, L4:FTFT)	[0, -8, 1]	no
(L0:TTTT, L4:TTFT)	[0, -8, 2, 1]	no
(L0:TTTT, L4:TTTT)	[0, -8, -8, 1]	no
(L0:TTTF, L4:TTFF)	[1, -7, 3, 0]	no
(L0:TTTF, L4:FTTF)	[0, -7, -8]	YES
(L0:TTTF, L4:TTTF)	[1, -7, -7, 0]	YES
(L0:TTFT, L7:TTFF)	[0, 2, -8, 1]	no
(L0:TTFT, L7:FTFT)	[0, 1, 2]	no
(L0:TTFT, L7:TTFT)	[0, 3, 1, 2]	no
(L0:TTFF, L0:TTTT)	[1, 2, -1, 0]	no

Fig. 6. The results of test generation for the running example

consisting of a sequence of $must^-$-transitions followed by one (and perhaps no) *may*-transition, while covering no state more than once. This can be done by a simple depth-first search procedure. The idea is that if we generate tests to cover these paths then we are guaranteed that the rest of the states in L will be covered if the execution of program does not go wrong (uncover an error).

In Figure 5, using such a depth-first search identifies ten paths. These ten paths through L are uniquely identified by their beginning and ending vertices, as shown in the column "**Path Endpoints**" in Figure 6.

6.2 Symbolic Execution

Each of the ten paths induces a straight-line C "path" program that we automatically generated by tracing the path through the `partition` function. Consider the path from `L0`:TTTF to the `L4`:TTFF:

`L0`:TTTF → `L1`:TTTF → `L2`:TTTF → `L3`:TTTF → `L4`:TTFF

and its corresponding path program (see Figure 7). There are four transitions between labels in this path. The transition `L0`:TTTF → `L1`:TTTF corresponds to the expression in **while** loop at label `L0` evaluating to true. This is modeled by the statement `assume(lo<=hi)` in the path program in Figure 7. The four statements corresponding to the four transitions are presented after the "prelude" code in Figure 7. The `assert` statement at the end of the path program asserts that the final state at label `L4` (TTFF) cannot occur, which of course is not true.

We used CBMC [CKY03], a bounded-model checker for C programs to generate a counterexample to the assertion that the state `L4`:TTFF cannot occur. CBMC produces an input array `a[]` and array length `n` that will cause the `assert` statement to fail, proving that `L4`:TTFF is reachable. For the generated path program of Figure 7, CBMC finds a counterexample and produces the input array [1, -7, 3, 0], as shown in the second column of Figure 6.

```
partition(int a[],int n) {
  assume(n>2);           // prelude
  pivot = a[0];          // prelude
  lo = 1;                // prelude
  hi = n-1;              // prelude

  assume(lo<=hi);        // L0:TTTF -> L1:TTTF
  ;                      // L1:TTTF -> L2:TTTF
  assume(a[lo]<=pivot); // L2:TTTF -> L3:TTTF
  lo=lo+1;               // L3:TTTF -> L4:TTFF

  assert(! ((lo<hi)&&(lo<=hi)&&
            !(a[lo]<=pivot)&&!(a[hi]>pivot))
         );
}
```

Fig. 7. The "path" program corresponding to the path L0:TTTF → L1:TTTF → L2:TTTF → L3:TTTF → L4:TTFF

6.3 Observe Test Runs

Instrumentation of the original program both collects the executed observable states for each test run and checks for array bounds violations. In our example, there are ten runs, two of which produce array bounds violations (because the `lo` index is incremented past the end of the input array and then `a[lo]` is accessed), as shown in the third column of Figure 6.

The set of observed states resulting from executing all ten tests contains all the states in Figure 5 except four of the states in $U - L$ (in particular, L5:FFFT, L6:FFFT and L7:FFFT and L3:FFTF) and the state L2:FFFF, which is unreachable due to an array bounds violation.

Fixing the error in the program and rerunning our entire process results in an upper bound U with 56 states and a lower bound L of 37 states. Test generation succeeds in covering all 37 states in the lower bound L and causes no array bounds errors. Additionally, these tests cover 6 of the 19 tests in $U - L$.

6.4 Abstraction Refinement

This leads us to consider whether or not the remaining 13 states in $U - L$ are reachable at all and to the problem of refining the upper and lower bounds. Consider the state L7:FFFT from Figure 5, which is in $U - L$ and was not covered by any test. The concretization of this abstract state is

```
lo>hi && a[lo]>pivot && a[hi]>pivot
```

Notice that `partition` function, while having an array bounds error, does correctly maintain the invariant that all array elements with index less than the variable `lo` have value less than or equal to `pivot`. However, in the above state, we have that `hi<lo` and `a[hi]>pivot`. Thus, it is not possible to reach this state.

We submit that rather than ignore abstract states whose concrete counterparts are unreachable, it is important to introduce new predicates to try and eliminate such states in the abstraction. The reason is that these unreachable states often will point to boundary conditions that have not yet been tested.

In order to eliminate the state `L7`:FFFT we will introduce three new predicates into the Boolean abstraction (in addition to the four already there) in order to track the status of the array when the variable `lo` takes on the value `hi+1`:

`(lo==hi+1)`, `(a[lo-1]<=pivot)`, `(a[hi+1]>pivot)`

These predicates track an important boundary condition that was not observed by the initial four predicates. With these additional predicates, the generated Boolean abstraction has matching lower and upper bounds ($L = U$) and our test generation process covers all reachable observable states. As mentioned before, we can not expect to be able to achieve matching lower and upper bounds in general. We will next consider what the condition $L = U$ means.

7 Discussion

To recap, U is the set of abstract states reachable (from the initial set of abstract states I_A) via a sequence of *may*-transitions, while L is the set of states reachable from I_A via a sequence of $must^-$-transitions, followed by a most one *may*-transition, followed by a sequence of $must^+$-transitions.

An abstract TTS M_A bisimulates [Mil99] a CTS M_C if each *may*-transition in M_A is matched by a $must^+$-transition (that is, $\xrightarrow{may}_A = \xrightarrow{must^+}_A$). It is easy to see that if M_A bisimulates M_C then every abstract state in U is reachable via a sequence of $must^+$-transitions. Bisimulation guarantees a *strong* form of reachability:

> if M_A bisimulates M_C and $s'_a \in reach[\xrightarrow{may}_A](\{s_a\})$ then **for all** concrete states $s_c \in \gamma(s_a)$, there exists $s'_c \in \gamma(s'_a)$ such that $s'_c \in reach_C(s_c)$

Thus, bisimulation implies that $L = U$.

However, if $L = U$ it does not follow that M_A bisimulates M_C because the definition of L permits a sequence of $must^-$-transitions, followed by a most one *may*-transition, followed by a sequence of $must^+$-transitions. The condition $L = U$ guarantees a *weak* form of reachability:

> if $L = U$ and $s'_a \in reach[\xrightarrow{may}_A](\{s_a\})$ then **there exists** concrete state $s_c \in \gamma(s_a)$ and there exists $s'_c \in \gamma(s'_a)$ such that $s'_c \in reach_C(s_c)$

So, the condition $L = U$ implies that there is a finite set of tests sufficient to observe all states in U. Since U is an upper bound to the set of reachable observable states R this set of tests covers all states in R as well (that is, $R = U = L$). From this it also follows that the condition $L = U$ is a sufficient test for determining the completeness of a *may*-abstraction [GRS00], since $L = U$ implies that $U = R$ and $R = \alpha(reach_C(I_C))$ is the most precise (complete) abstract answer possible.

In other words, if $U = L$ then the set U is equal to the set of observable states (R) that would be encountered during the (infinite) computation of the least fixpoint over the concrete transition system M_C, represented by $\alpha(reach_C(I_C))$.

To summarize, the condition $U = L$ joins together the worlds of testing and abstraction. It implies both a sound and complete abstract domain that can be completely covered by a finite set of tests.

8 Related Work

Related work breaks into a number of topics.

8.1 Control-Flow Coverage Criteria

We have already compared PCT coverage with statement, branch, multiple condition, predicate and path coverage (see Section 2). We now consider other alternatives to path coverage, namely linear code sequence and jump (LCSAJ) coverage and data-flow coverage based on def-use pairs. An LCSAJ represents a sequence of statements (which may contain conditional statements) ending with a branch. An LCSAJ is an acyclic path (no edge appears twice) through a control-flow graph ending with a branch statement. As we have shown, PCT coverage is incomparable to path coverage for loop-free programs, so it also is incomparable to LCSAJ coverage. The goal of def-use coverage is to cover, for each definition d of a variable x and subsequent use u of variable x, a path from d to u not containing another definition of x. If there is such a path from d to u then there is an acyclic path from d to u that doesn't contain another definition of x, so again PCT coverage is incomparable to def-use coverage.

8.2 Symbolic Execution and Test Generation

The idea of using paths and symbolic execution of paths to generate tests has a long and rich history going back to the mid-1970's [BEL75, How76, Cla76, RHC] and continuing to the present day [JBW$^+$94, GBR98, GMS98]. Recently, Chlipala et al. proposed using counterexample-driven refinement to guide test generation [CHJM04]. The major contribution of our work over previous efforts in this area is to guide test generation using Boolean abstraction and the computation of upper and lower bounds to the set of reachable observable states.

A classic problem in path-based symbolic execution is the selection of program paths. One way to guide the search for feasible paths is to execute the program symbolically along all paths, while guiding the exploration to achieve high code coverage. Clearly, it is not possible to symbolically execute all paths, so the search must be cut off at some point. Often, tools will simply analyze loops through one or two iterations [BPS00]. Another way to limit the search is to bound the size of the input domain (say, to consider arrays of at most length three) [JV00], or to bound the maximum path length that will be considered, as done in bounded model checking [CKY03]. An experiment by Yates

and Malevris provided evidence that the likelihood that a path is feasible decreases as the number of predicates in the path increases [YM89]. This led them to use shortest-path algorithms to find a set of paths that covers all branches in a function.

In contrast to all these methods, our technique uses the set of input predicates to bound the set of paths that will be used to generate test data for a program. The predicates induce a Boolean abstraction that guides the selection of paths.

Other approaches to test generation rely on dynamic schemes. Given an existing test t, Korel's "goal-oriented" approach seeks to perturb t to a test t' cover a particular statement, using function minimization techniques [Kor92]. The potential benefit of Korel's approach is that it is dynamic and has an accurate view of memory and flow dependences. The downside of his approach is that test t may be very far away from a suitable test t'.

Another dynamic approach to test generation is found in the Korat tool [BKM02]. This tool uses a function's precondition on its input to automatically generate all (nonisomorphic) test cases up to a given small size. It exhaustively explores the input space of the precondition and prunes large portions of the search space by monitoring the execution of the precondition. For an example such as the `partition` function that has no constraints on its input, the Korat method may not work very well. Furthermore, it requires the user to supply a bound on the input size whereas our technique infers the input size.

Harder, Mellen and Ernst [HME03] propose using operational abstractions (properties inferred from observing a set of test executions) to guide the generation and maintenance of test suites. This is similar in spirit to predicate-complete testing but unsound (the properties inferred are "likely" invariants but not guaranteed to hold in general). In contrast, our use of predicate abstraction and reachability analysis in the abstract domain computes a (sound) overapproximation to the set of reachable observable states of a program. Furthermore, the invariants we can establish about a program's behavior involve arbitrary Boolean expressions over atomic predicates whereas Harder et al. limit themselves to atomic predicates and implications between atomic predicates.

8.3 Three-Valued Model Checking

Our work was inspired by the work on three-valued model checking by Bruns, Godefroid, Huth and Jagadeesan [BG99, GHJ01, GR03]. Their work shows how to model incomplete (abstract) systems using modal transition systems (equivalently, partial Kripke Structures), as we have done here. It then gives algorithms for model checking temporal logic formula with respect to such systems. Given an MTS, these algorithms can determine whether a temporal logic formula is definitely true, definitely false or unknown with respect to the MTS.

Our computation of lower and upper bounds achieves a similar result but infers reachability properties of a concrete TTS M_C from analysis of an abstract TTS M_A. The lower bound L characterizes those observable states that are definitely reachable, the upper bound U (more precisely, its inverse $S - U$)

characterizes those observable states that are definitely not reachable, and the reachability status of states in $U - L$ are unknown.

To achieve a more precise lower bound for (weak) reachability, we generalized the definition of *must*-transitions given for MTS to account for three types of *must*-transitions: $must^+$ (which correspond to *must*-transitions in an MTS), $must^-$ and $must^\#$.

In model checking of abstractions of concrete transition systems, one is interested in proving that a temporal property holds for *all* concrete execution paths starting from some initial abstract state. This is the reason why only $must^+$-transitions are used in model checking of modal transitions systems. For (weak) reachability, one is interested proving the existence of *some* concrete execution path starting from some initial abstract state. Thus, $must^-$-transitions are of interest.

9 Conclusion

We have presented a new form of control-flow coverage that is based on observing the vector consisting of a program's conditional predicates, thus creating a finite-state space. There are a number of open questions to consider. First, what is a logical characterization of tri-modal transition systems? Second, how can one automate the refinement process to bring the lower and upper bounds closer? (It is well known that the set of *must*-transitions is not generally monotonically non-decreasing when predicates are added to refine an abstract system. Recently, Shoham and Grumberg [SG04] and Alfaro, Godefroid and Jagadeesan [dAGJ04] independently proposed a new form of *must*-transition that permits monotonic refinement of abstractions.) Finally, how does this technique work in practice?

Acknowledgements

Thanks to Daniel Kroening for his help with the CBMC model checker. Thanks also to Byron Cook, Tony Hoare, Vladimir Levin, Orna Kupferman and Andreas Podelski for their comments.

References

[BEL75] R. Boyer, B. Elspas, and K. Levitt. SELECT–a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Notices*, 10(6):234–245, 1975.

[BG99] G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *CAV 99: Computer Aided Verification*, LNCS 1633, pages 274–287. Springer-Verlag, 1999.

[BKM02] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 123–133. ACM, 2002.

[BPS00] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, June 2000.

[CHJM04] A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In *ICSE 04: International Conference on Software Engineering (to appear)*. ACM, 2004.

[CKY03] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Design Automation Conference*, pages 368–371, 2003.

[Cla76] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, September 1976.

[dAGJ04] L. de Alfaro, P. Godefroid, and R. Jagadeesan. Three-valued abstractions of games:uncertainty, but with precision. In *LICS 04: Logic in Computer Science*, To appear in LNCS. Springer-Verlag, 2004.

[GBR98] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 53–62. ACM, 1998.

[GHJ01] P. Godefroid, Michael Huth, and Radha Jagadeesan. Abstraction-based model checking using modal transition systems. In *CONCUR 01: Conference on Concurrency Theory*, LNCS 2154, pages 426–440. Springer-Verlag, 2001.

[GMS98] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *FSE 98: Foundations of Software Engineering*. ACM, 1998.

[God03] P. Godefroid. Reasoning about abstract open systems with generalized module checking. In *EMSOFT 03: Conference on Embedded Software*, LNCS 2855, pages 223–240. Springer-Verlag, 2003.

[GR03] P. Godefroid and R.Jagadeesan. On the expressiveness of 3-valued models. In *VMCAI 03: Verification, Model Checking and Abstract Interpretation*, LNCS 2575, pages 206–222. Springer-Verlag, 2003.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[GRS00] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.

[HME03] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *ICSE 2003: International Conference on Software Engineering*, pages 60–71. ACM, 2003.

[How76] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2:208–215, 1976.

[JBW+94] R. Jasper, M. Brennan, K. Williamson, B. Currier, and D. Zimmerman. Test data generation and feasible path analysis. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 95–107. ACM, 1994.

[JV00] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 14–25. ACM, 2000.

[Kor92] B. Korel. Dynamic method of software test data generation. *Software Testing, Verification and Reliability*, 2(4):203–213, 1992.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[RHC] C. Ramamoorthy, S. Ho, and W. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293–300.

[SG04] S. Shoham and O. Grumberg. Monotonic abstraction-refinement for CTL. In *TACAS 04: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2988, pages 546–560. Springer-Verlag, 2004.

[Tai96] K-C. Tai. Theory of fault-based predicate testing for computer programs. *IEEE Transactions on Software Engineering*, 22(8):552–562, 1996.

[Tai97] K-C. Tai. Predicate-based test generation for computer programs. In *ICSE 97: International Conference on Software Engineering*, pages 267–276, 1997.

[YM89] D. Yates and N. Malevris. Reducing the effects of infeasible paths in branch testing. In *Proceedings of the Symposium on Software Testing, Analysis, and Verification*, pages 48–54. ACM, 1989.

A Perspective on Component Refinement

Luís S. Barbosa

Universidade do Minho, DI - CCTC, Campus de Gualtar,
4710-057 Braga, Portugal
lsb@di.uminho.pt

Abstract. This paper provides an overview of an approach to coalgebraic modelling and refinement of state-based software components, summing up some basic results and introducing a discussion on the interplay between behavioural and classical data refinement. The approach builds on coalgebra theory as a suitable tool to capture observational semantics and to base an abstract characterisation of possible behaviour models for components (from partiality to different degrees of non-determinism).

1 Introduction

In recent years *component-based software development* [46,49] emerged as a promising paradigm to deal with the ever increasing need for mastering complexity in software design, evolution and reuse. However, as it happened before with object-orientation, and software engineering in the broad sense, component-orientation has grown up to a collection of popular technologies, methods and tools, before consensual definitions and principles (let alone formal foundations) have been put forward.

This paper focus on a particular corner of the 'componentware' landscape. A corner in which software components are regarded as specifications of *state-based* modules, in the tradition of the so-called *model oriented* approach to formal systems design — a widespread paradigm of which VDM [23], Z [45], B [1] and RAISE [47] are well-known representatives. In a series of papers, starting with [6] and including [8,7,9], a coalgebraic characterisation of this sort of components and a corresponding calculus was proposed. This approach defines components as persistent units which encapsulate a number of services through public interfaces and provide limited access to internal state spaces. Coalgebra theory [42] was found a suitable tool to capture observational semantics and to base an abstract characterisation of possible *behaviour models* for components (*e.g.*, partiality or (different degrees of) non-determinism). Such models are introduced in the framework in a *generic* [5] way — *i.e.*, as a *parameter*, in the form of a strong monad in the component calculus. More recently in [29,30] the framework was extended from an equivalence to a refinement calculus, based on a weak form of coalgebra morphism.

This paper provides an overview of this approach to component modelling and refinement, summing up some basic results and introducing a discussion

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 23–48, 2005.
© Springer-Verlag Berlin Heidelberg 2005

on the interplay between behavioural refinement, as introduced in [29,30], and classical *data* refinement [18,35] applied to software components.

Section 2 motivates the use of coalgebras in component modelling and reviews the component calculus introduced in [8,7]. This paves the way for a detailed discussion of component refinement at both the *behavioural* and *data* levels in sections 3 and 4, respectively. Finally, section 5 introduces some recent research concerns on this topic.

2 Coalgebraic Models for Software Components

2.1 Coalgebras

One of the most elementary models of a software component, or of any computational process whatsoever, is that of a *function*

$$f : O \longleftarrow I$$

which specifies a deterministic transformation rule between two structures I and O. In a (metaphorical) sense, this may be dubbed as the 'engineer's view' of reality: *here is a recipe (a tool, a technology) to build gnus from gnats.*

Often, however, reality is not so simple. For example, one may know how to produce 'gnus' from 'gnats' but not in all cases. This is expressed by observing the output of f in a more refined context: O is replaced by $O + \mathbf{1}$, where $\mathbf{1}$ denotes the singleton datatype and $+$ is datatype sum or coproduct. Then f is said to be a *partial* function. In other situations one may recognise that there is some environmental (or context) information about 'gnats' that, for some reason, should be hidden from input. It may be the case that such information is too extensive to be supplied to f by its user, or that it is shared by other functions as well. It might also be the case that building gnus would eventually modify the environment, thus influencing latter production of more 'gnus'. For U a denotation of such context information, the signature of f becomes[1]

$$f : (O \times U)^U \longleftarrow I$$

A function computed within a context is often referred to as 'state-based', in the sense the word 'state' has in automaton theory — the internal memory of the automaton which both constraints and is constrained by the execution of actions. In fact, the 'nature' of f as a 'state-based function' is made more explicit by rewriting its signature as

$$f : (O \times U)^I \longleftarrow U$$

This, in turn, may suggest an alternative computational model, which (again in a metaphorical sense) one may dub as the 'natural scientist's view'. Instead of

[1] In the sequel we often adopt the standard mathematical notation B^A for funtional dependency, instead of the equivalent $[A \to B]$ more familiar in computing.

a recipe to build 'gnus' from 'gnats', we are left with the awareness that *there exist gnus and gnats and that their evolution can be observed*.

The able 'natural scientist' will equip herself with the right 'lens' — that is, a tool to observe with, which necessarily entails a particular *shape* for observation. The basic ingredients required to support such an 'observational', or 'state-based', view of computational processes may be summarised as follows:

a *lens*:		(a functor T)
an *observation structure*:	universe $\xleftarrow{p}$ universe	(a T-coalgebra)

Technically, in the category Set of sets and set-theorectical functions, a *coalgebra* for a functor T is a set U, which corresponds to the object being observed (the *carrier*), and a function $p : \mathsf{T}\, U \longleftarrow U$ [2].

There is, of course, a great diversity of 'lenses' and, for the same 'lens', a variety of observation structures, *i.e.*, of coalgebras. Moreover, such structures can be related and compared. This entails the need for a notion of *homomorphism*, *i.e.*, a map which preserves the shape of T as an observation tool. Therefore, a T-coalgebra morphism h between, say, coalgebras p and q is just a function between the respective carriers making the following diagram to commute:

$$\begin{array}{ccc} U & \xrightarrow{p} & \mathsf{T}\, U \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle \mathsf{T}\, h} \\ V & \xrightarrow[q]{} & \mathsf{T}\, V \end{array}$$

Let us consider some possible lenses. An extreme case is the opaque lens: no matter what we try to observe through it, the outcome is always the same. Formally, such a lens is the constant functor $\underline{\mathbf{1}}$ which maps every object to the singleton set $\mathbf{1}$ and every morphism to the identity on $\mathbf{1}$. Since $\mathbf{1}$ is the final object in Set, all $\mathbf{1}$-coalgebras reduce to ! — the canonical function to $\mathbf{1}$. A slightly more interesting lens is $\underline{\mathbf{2}}$, which allows states to be classified into two different classes: black or white. This makes it possible to identify *subsets* of the 'universe' U under observation, as an observation structure p for this functor

[2] The *dual* perspective emphasises the possibility of at least some (essentially finite) things being not only observed, but actually *built*. In this case, one does not work with a 'lens' but with a 'toolbox'. The *assembly process* is specified in a similar (but dual) way:

a *tool box*:		(a functor T)
an *assembly process*:	artifact $\xleftarrow{d}$ artifact	(a T-algebra)

maps elements of U to one or another element of $\mathbf{2}$. Should an arbitrary set O be chosen to colour the lens, the possible observations become more discriminating. A coalgebra for $\underline{O}$ is a 'colouring' device in the sense that elements of the universe are classified (*i.e.*, regarded as distinct) by being assigned to different elements of O.

Thus, a 'colour set' as $\mathbf{1}$, $\mathbf{2}$ or O above, can be regarded as a *classifier* of the state space. Coalgebras, for such constant functors, are *pure* observers providing a limited access to the state space by mapping it into the 'colour set' — or *attributes*, as they are known in object-oriented programming.

A common assumption on state-based components is that the state itself is a 'black box': it may evolve either internally or as a reaction to external stimuli, but the only way of tracing such an evolution is by observing the values of its attributes. Under this assumption the 'transparent' lens is not particularly useful. Technically, it corresponds to the *identity* functor Id. An observation structure for Id amounts to a function $p : U \longleftarrow U$. This means that, by using p, the state U can indeed be modified, an ability we hadn't before. But, on the other hand, the absence of attributes makes any meaningful observation impossible. The best we can say, if no direct access to U is allowed, is just that *things happen*. A better alternative is to combine attributes with such state modifiers, or update operations, to model the 'universe' evolution. The latter will be called *actions* here; in the object paradigm they are known as *methods*. Such a combination leads to a richer stock of lens. We might consider, for example, that

- *things happen and disappear*: $\mathsf{T}\ U\ =\ U + \mathbf{1}$
- *things happen and, in doing so, some of their attributes become visible*, *i.e.*, (non trivial) output is produced: $\mathsf{T}\ U\ =\ U \times O$
- additional *input* is required for an observation to take place: $\mathsf{T}\ U\ =\ U^I$
- *we are not completely sure about what has happened*, in the sense that the evolution of the system being observed may be nondeterministic. In this case, the lens above can be combined with $\mathsf{T}\ U\ =\ \mathcal{P}U$ where $\mathcal{P}U$ is the finite powerset of U.

In the second example, the action also has an *input interface*. Typically, actions over the same state space cannot happen simultaneously and, therefore, if more than one is specified in a particular structure, in each execution the input supplied will also select the action to be activated. In some cases, the input is there only for selection purposes: actions with trivial input (*i.e.*, $I = \mathbf{1}$) correspond to buttons that can be pressed. Then the input interface organises itself as a coproduct. Attributes, on the other hand, can be inspected in parallel. In other cases still we might be intereseted in methods which not only change the internal state of a component but also produce an observable output. Putting all the ingredients together we arrive at the following functor as a possible shape for software components modelled as coalgebras:

$$\mathsf{T}\ =\ A \times (\mathsf{Id} \times O)^I \tag{1}$$

where $A = \Pi_{k \in K} A_k$ stand for the product of the attribute types and $I = \sum_{j \in J} I_j$ and $O = \sum_{j \in J} O_j$ correspond to the coproduct of, respectively, input and output parameters of the component operations.

Functor T can still be enriched with a specification of a particular *behavioural model* to which components may stick to. Notice the use of the *maybe* or the *powerset* monads above to capture such models. Therefore functor T in (1) becomes parametric on an arbitrary *strong monad*[3] B, leading to coalgebras for

$$\mathsf{T} \ = A \times \mathsf{B}(\mathsf{Id} \times O)^I \tag{2}$$

as a possible general model for state based software components. Therefore computation of an action will not simply produce an output and a continuation state, but a B-structure of such pairs. The monadic structure provides tools to handle such computations. Unit (η) and multiplication (μ), provide, respectively, a value embedding and a 'flatten' operation to reduce nested behavioural effects. Strength, either in its right (τ_r) or left (τ_l) version, cater for context information.

Several possibilities can be considered for B. The simplest case is, obviously, the *identity* monad, Id, whereby components behave in a totally *deterministic* way. Other possibilities capturing more complex behavioural features, include the maybe monad ($\mathsf{B} = \mathsf{Id} + \mathbf{1}$) for *partiality*, the (finite) powerset ($\mathsf{B} = \mathcal{P}$) or sequence ($\mathsf{B} = \mathsf{Id}^*$) monads for (arbitrary or ordered) *non determinism* or the *bag monad* [4] to model cases in which among the possible future evolutions of a component, some are stipulated to be more likely (cheaper, more secure, *etc.*) than others (see [8] for further details on the use of monads in a calculus of generic behaviour models).

2.2 Components

Building on the discussion above, this subsection introduces component's specifications as coalgebras and gives a glimpse of the resulting calculus. Without a major loss of generality, however, we shall concentrate in this text on coalgebras for

$$\mathsf{T} \ = \ \mathsf{B}(\mathsf{Id} \times O)^I \tag{3}$$

[3] A *strong monad* is a monad $\langle \mathsf{B}, \eta, \mu \rangle$ where B is a strong functor and both η and μ are strong natural transformations [26]. B being strong means there exist natural transformations $\mathsf{T}(\mathsf{Id} \times -) : \mathsf{T} \times - \Longleftarrow \mathsf{T} \times -$ and $\mathsf{T}(- \times \mathsf{Id}) : - \times \mathsf{T} \Longleftarrow - \times \mathsf{T}$, called the right and left strength, respectively, subject to certain conditions. Their effect is to *distribute* the free variable values in the context "$-$" along functor B. Strength τ_r, followed by τ_l maps $\mathsf{B}I \times \mathsf{B}J$ to $\mathsf{BB}(I \times J)$, which can, then, be flattened to $\mathsf{B}(I \times J)$ via μ. In most cases, however, the *order* of application is relevant for the outcome. The Kleisli composition of the right with the left strength, gives rise to a natural transformation whose component on objects I and J is given by $\delta_r = \tau_{r_{I,J}} \bullet \tau_{l_{\mathsf{B}I,J}}$ Dually, $\delta_l = \tau_{l_{I,J}} \bullet \tau_{r_{I,\mathsf{B}J}}$. Such transformations specify how the monad distributes over product and, therefore, represent a sort of sequential composition of B-computations.

[4] Defined over a structure $\langle M, \oplus, \otimes \rangle$, where both $\oplus$ and $\otimes$ are Abelian monoids, the latter distributing over the former.

therefore ommiting the attribute's part in (2). Notice that, for $\mathsf{B} = \mathsf{Id}$, such coalgebras correspond to classical *Mealy* machines [27]. In general a *component specification* is defined as follows, where a collection of sets I, O, ..., acting as component interfaces is assumed.

Definition 1. *A software component is specified by a pointed coalgebra*

$$\langle u_p \in U_p, \overline{a}_p : \mathsf{B}(U_p \times O)^I \longleftarrow U_p \rangle \tag{4}$$

where u_p is the initial state, often referred to as the seed *of the component computation, and the coalgebra dynamics is captured by currying a state-transition function $a_p : \mathsf{B}\ (U_p \times O) \longleftarrow U_p \times I$.*

An elementary, but typical, example of a state based component is given by the following specification of a buffering device which provides services to store and deliver messages.

Example 1. *Denoting by U its internal state, a buffer for messages of type M is handled through operations*

$$\mathsf{put} : U \longleftarrow U \times M$$
$$\mathsf{pick} : U \times M \longleftarrow U$$

An alternative, 'black box' view hides U from the component's environment and regards each operation as a pair of input/output ports. Such a 'port' signature of, e.g.*, the* pick *operation is given by*

$$\mathsf{pick} : M \longleftarrow \mathbf{1}$$

The intuition is that pick *is activated with the simple pushing of a 'button' (its argument being the buffer private state space) whose effect is the production of a M value in the corresponding output port. Similarly typing* put *as*

$$\mathsf{put} : \mathbf{1} \longleftarrow M$$

means that an external argument is required on activation but no visible output is produced, but for a trivial indication of successful termination. Such 'port' signatures are grouped together in the diagram below. Note how input (respectively, output) 'ports' are represented by the sum of their parameters. Such sums label the buffer input (respectively, output) point represented by an empty (respectively, full) circle in the diagram. Combined input type $M+\mathbf{1}$ models the choice between the two functionalities.
One might capture Store *dynamics by a function $a_{\mathsf{Store}} : \mathcal{P}(U \times O) \longleftarrow U \times I$ which describes how* Store *reacts to input stimuli, produces output data (if any) and changes state. This can also be written in a curried form as $\overline{a}_{\mathsf{Store}} : \mathcal{P}(U \times O)^I \longleftarrow U$ that is, as a* coalgebra *of signature $U \longleftarrow \mathsf{T}\ U$ where functor T captures transition 'shape':*

$$\mathsf{T} = \mathcal{P}(\mathsf{Id} \times O)^I \tag{5}$$

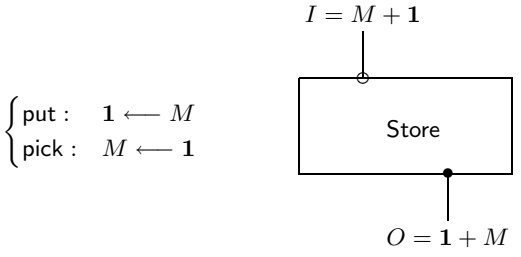


Fig. 1. The Store component

Built in this 'shape' is the possibility of non deterministic evolution captured by the finite powerset monad. Concretely, our first model for Store *given below assumes that messages are labelled by a time tag (provided by a clock function* ttag()*) so that on the arrival of a* pick *request any message stored for more than a specified time interval (ϵ) can be delivered. Let $U = \mathcal{P}(M \times T)$, where T stands for a suitable representation of time, be its state space. Then,*

$$\begin{aligned} a_{\mathsf{Store}}\langle u, \mathsf{put}\ m\rangle &= \{\langle u \cup \{\langle m, \mathsf{ttag}()\rangle\}, \iota_1\ *\rangle\} \\ a_{\mathsf{Store}}\langle u, \mathsf{pick}\rangle &= \{\langle u \setminus \{\langle m, t\rangle\}, \iota_2\ m\rangle |\ \langle m, t\rangle \in u \wedge t - \mathsf{ttag}() \geq \epsilon\} \end{aligned}$$

where put m *and* pick *abbreviate $\iota_1\ m$ and $\iota_2\ *$, respectively.*

Components can be regarded as arrows between (input/output) interfaces and therefore arrows between components are arrows between arrows. Formally, these three notions — *interfaces*, *components* and *component morphisms* — lead to the notion of a *bicategory* [5] as a possible mathematical universe for components to live. In brief, we take interfaces as *objects* of a bicategory Cp, whose *arrows* are pointed T-coalgebras and *2-cells*, the arrows between arrows, the corresponding morphisms. Formally,

Definition 2. *Assume arbitrary sets as* Cp *objects. For each pair $\langle I, O\rangle$ of objects, define a category* $\mathsf{Cp}(I, O)$*, whose arrows*

$$h : \langle u_q, \overline{a}_q\rangle \longleftarrow \langle u_p, \overline{a}_p\rangle \qquad \begin{array}{ccc} U_p & \xrightarrow{\overline{a}_p} & \mathsf{T}\ U_p \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle \mathsf{T}^{\mathsf{B}}\ h} \\ U_q & \xrightarrow[\overline{a}_q]{} & \mathsf{T}\ U_q \end{array}$$

[5] Basically a *bicategory* [11] is a category in which a notion of arrows between arrows is additionally considered. This means that the the space of morphisms between any given pair of objects, usually referred to as a (hom-)set, acquires itself the structure of a category. Therefore arrow composition and unit laws become functorial, since they transform both objects and arrows of each hom-set in an uniform way.

satisfy the following morphism *and* seed preservation *conditions:*

$$\overline{a}_q \cdot h \;=\; \mathsf{T}\, h \cdot \overline{a}_p \tag{6}$$

$$h\; u_p \;=\; u_q \tag{7}$$

Composition is inherited from Set *and the identity* $1_p : p \longleftarrow p$*, on component* p*, is defined as the identity* id_{U_p} *on the carrier of* p*. Next, for each triple of objects* $\langle I, K, O\rangle$*, a composition law is given by a functor*

$$;_{I,K,O} : \mathsf{Cp}(I,O) \longleftarrow \mathsf{Cp}(I,K) \times \mathsf{Cp}(K,O)$$

whose action on objects p *and* q *is given by*

$$p\,;\,q \;=\; \langle \langle u_p, u_q\rangle \in U_p \times U_q, \overline{a}_{p;q}\rangle$$

where $a_{p;q} : \mathsf{B}(U_p \times U_q \times O) \longleftarrow U_p \times U_q \times I$ *is detailed as follows*

$$\begin{aligned} a_{p;q} = {} & U_p \times U_q \times I \xrightarrow{\;\cong\;} U_p \times I \times U_q \xrightarrow{\;a_p \times \mathsf{id}\;} \mathsf{B}(U_p \times K) \times U_q \\ & \xrightarrow{\;\tau_r\;} \mathsf{B}(U_p \times K \times U_q) \xrightarrow{\;\cong\;} \mathsf{B}(U_p \times (U_q \times K)) \\ & \xrightarrow{\;\mathsf{B}(\mathsf{id} \times a_q)\;} \mathsf{B}(U_p \times \mathsf{B}(U_q \times O)) \xrightarrow{\;\mathsf{B}\tau_l\;} \mathsf{BB}(U_p \times (U_q \times O)) \\ & \xrightarrow{\;\cong\;} \mathsf{BB}(U_p \times U_q \times O) \xrightarrow{\;\mu\;} \mathsf{B}(U_p \times U_q \times O) \end{aligned}$$

The action of ń;ż on 2-cells reduces to $h\,;\,k \;=\; h \times k$*. Finally, for each object* K*, an identity law is given by a functor*

$$\mathsf{copy}_K : \mathsf{Cp}(K,K) \longleftarrow \mathbf{1}$$

whose action on objects is the constant component $\langle * \in \mathbf{1}, \overline{a}_{\mathsf{copy}_K}\rangle$*, where* $a_{\mathsf{copy}_K} = \eta_{\mathbf{1}\times K}$*. Slightly abusing notation, this will be also referred to as* copy_K*. Similarly, the action on morphisms is the constant morphism* $\mathsf{id}_{\mathbf{1}}$*.*

2.3 A Component Calculus

The fact that, for each strong monad B, components form a bicategory amounts not only to a standard definition of two basic combinators ; and copy_K of a possible component calculus, but also to setting up its laws in the form of bisimulation equations. Therefore, the existence of a seed preserving morphism between two components makes them T^{B}-bisimilar, leading to the following laws, for appropriately typed components p, q and r:

$$\mathsf{copy}_I\,;\,p \;\sim\; p \;\sim\; p\,;\,\mathsf{copy}_O \tag{8}$$

$$(p\,;\,q)\,;\,r \;\sim\; p\,;\,(q\,;\,r) \tag{9}$$

In previous papers [8,7,9] we have proposed an algebra of T-components parametric on a behaviour model B. The development of such a calculus starts from the simple observation that functions can be regarded as particular instances of components, whose interfaces are given by their domain and codomain types. Formally,

Definition 3. *A function* $f : B \longleftarrow A$ *is represented in* Cp *by*

$$\ulcorner f \urcorner \;=\; \langle * \in \mathbf{1}, \overline{a}_{\ulcorner f \urcorner} \rangle$$

i.e., *a coalgebra over* $\mathbf{1}$ *whose action is given by the currying of*

$$a_{\ulcorner f \urcorner} = \mathbf{1} \times A \xrightarrow{\mathsf{id} \times f} \mathbf{1} \times B \xrightarrow{\eta_{(\mathbf{1} \times B)}} \mathsf{B}(\mathbf{1} \times B)$$

The pre- and post-composition of a component with Cp-lifted functions can be encapsulated into an unique combinator, called *wrapping*, which is reminiscent of the *renaming* connective found in process calculi (*e.g.*, [31]). Let $p : O \longleftarrow I$ be a component and consider functions $f : I \longleftarrow I'$ and $g : O' \longleftarrow O$. Component p wrapped by f and g, denoted by $p[f, g]$ and typed as $O' \longleftarrow I'$, is defined by input pre-composition with f and output post-composition with g. Formally,

Definition 4. *The* wrapping *combinator is a functor*

$$-[f, g] : \mathsf{Cp}(I', O') \longleftarrow \mathsf{Cp}(I, O)$$

which is the identity on morphisms and maps component $\langle u_p, \overline{a}_p \rangle$ *into* $\langle u_p, \overline{a}_{p[f,g]} \rangle$*, where*

$$a_{p[f,g]} = U_p \times I' \xrightarrow{\mathsf{id} \times f} U_p \times I \xrightarrow{a_p} \mathsf{B}(U_p \times O) \xrightarrow{\mathsf{B}(\mathsf{id} \times g)} \mathsf{B}(U_p \times O')$$

Three tensor products are also introduced in the calculus to model *choice* ($\boxplus$), *concurrent* ($\circledast$) and *parallel* composition ($\boxtimes$). The latter is detailed below for illustration purposes.

Parallel composition, denoted by $p \boxtimes q$, corresponds to a synchronous product: both components are executed simultaneously when triggered by a pair of legal input values. The behavioural effect, captured by monad B, propagates. For example, if B expresses component failure and one of the arguments fails, product fails as well. Formally,

Definition 5. *The* parallel *combinator* $\boxtimes$ *is defined as* $I \boxtimes J \;=\; I \times J$ *on objects and a family of functors*

$$\boxtimes_{IOJR} : \mathsf{Cp}(I \times J, O \times R) \longleftarrow \mathsf{Cp}(I, O) \times \mathsf{Cp}(J, R)$$

which yields

$$p \boxtimes q \;=\; \langle \langle u_p, u_q \rangle \in U_p \times U_q, \overline{a}_{p \boxtimes q} \rangle$$

where

$$\begin{aligned} a_{p \boxtimes q} \qquad & U_p \times U_q \times (I \times J) \xrightarrow{\cong} U_p \times I \times (U_q \times J) \\ & \xrightarrow{a_p \times a_q} \mathsf{B}\,(U_p \times O) \times \mathsf{B}\,(U_q \times R) \\ & \xrightarrow{\delta_l} \mathsf{B}\,(U_p \times O \times (U_q \times R)) \\ & \xrightarrow{\cong} \mathsf{B}\,(U_p \times U_q \times (O \times R)) \end{aligned}$$

and maps every pair of arrows $\langle h_1, h_2 \rangle$ *into* $h_1 \times h_2$*.*

A generic form of component interaction is achieved by a generalization of sequential composition, leading to a family of *hook* combinators which forces *part* of the output of a component to be fed back as input. For components with the same input/output type, the *hook* combinator has a particularly simple definition as the Kleisli composition of the original coalgebra. Formally,

Definition 6. *Let* $p : Z \longleftarrow Z$. *Define*

$$p \upharpoonleft : Z \longleftarrow Z \;= \langle u_p \in U_p, \overline{a}_{p\upharpoonleft} \rangle$$

where

$$a_{p\upharpoonleft} \;=\; U_p \times Z \xrightarrow{a_p} \mathsf{B}(U_p \times Z) \xrightarrow{\mathsf{B}a_p} \mathsf{BB}(U_p \times Z) \xrightarrow{\mu} \mathsf{B}(U_p \times Z)$$

i.e., $a_{p\upharpoonleft} = a_p \bullet a_p$,

The following example illustrates the use of some component combinators to connect elementary state-based specifications. The component to be built is known as the *game of life*, a simple model of cellular behaviour which has been popularised as a common screen locker for computers.

Example 2. *The game is based on a grid of cells each of which sends and receives elementary stimulus to and from its four adjacent neighbours. A stimulus is a Boolean value indicating whether the cell is either 'alive' or 'dead'. The following few rules govern the survival, death and birth of cell generations:*

- *Each living cell with less than two or more than three living neighbours dies in the next generation.*
- *Each dead cell with exactly three living neighbours becomes alive.*
- *Each living cell with less than two or three living neighbours survives until the next generation.*

Each cell will be specified as a component Cell *whose input is a tuple of four Boolean values, each one to be supplied by one of the four adjacent cells. The cell reacts to such a stimulus by computing its new state — 'dead' or 'alive' — and by making it available as an output to its neighbours, used to compute the next cell generation. Formally, we define*

$$\mathsf{Cell} : \mathbf{2} \longleftarrow \mathbf{2} \times \mathbf{2} \times \mathbf{2} \times \mathbf{2} \quad = \quad \langle \mathsf{true} \in \mathbf{2}, \overline{a}_{\mathsf{Cell}} \rangle$$

where

$$a_{\mathsf{Cell}}\, \langle u, t \rangle \;=\; \text{let } n = \mathsf{living}\, t \text{ in } \begin{cases} \langle \mathsf{false}, \mathsf{false} \rangle & \text{if } u = \mathsf{true} \wedge (n < 1 \vee n > 3) \\ \langle \mathsf{true}, \mathsf{true} \rangle & \text{if } u = \mathsf{false} \wedge n = 3) \\ \langle u, u \rangle & \text{otherwise} \end{cases}$$

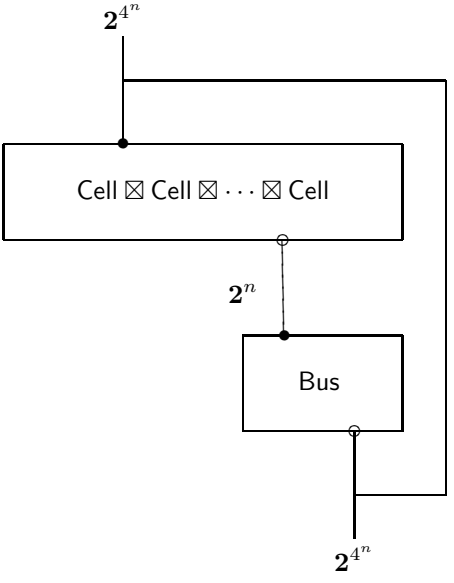
Function living *above, counts the number of living stimuli (*i.e., *the number of* true *values) in a four Boolean tuple. So,* $U_{\mathsf{Cell}} = \mathbf{2}$ *and* $\mathsf{B} = \mathsf{Id}$. *The game's*

behaviour is, of course, deterministic and all cells in the grid react simultaneously to produce the new generation. To form a grid of n cells we simply connect them using the parallel *combinator* $\boxtimes$. *The crucial point is to devise a wiring scheme to guarantee that the joint output of the n connected cells is appropriately fed back. The composed system is pictured below, where component*

$$\mathsf{Bus} : \mathbf{2}^{4^n} \longleftarrow \mathbf{2}^n$$

concentrates and correctly distributes the output.

The n cells are organised as a fully connected matrix of k rows and l columns ($n = k \times l$), *so that the neighbours of cell* $\langle i, j\rangle$ *are* $\langle i-1, j\rangle$, $\langle i+1, j\rangle$, $\langle i, j-1\rangle$ *and* $\langle i, j+1\rangle$ *(in the 'west', 'east', 'north' and 'south' directions, respectively) computed in the k and l rings (*i.e., $1-1=k$, $k+1=1$ *and* $1-1=l$, $l+1=1$*).*



To specify Bus *we adopt the following convention: the first cell in the* $\boxtimes$*-expression has coordinates* $\langle 1, 1\rangle$, *second is* $\langle 1, 2\rangle$ *and so on until column n is reached; the next cell is then* $\langle 2, 1\rangle$. *Under this convention the output produced by cell* $\langle i, j\rangle$ *is selected from the global output tuple as the* $j + (n \times (i-1))$*-projection,* i.e.

$$\begin{aligned} \mathsf{out}_{\langle i,j\rangle} &: \mathbf{2}^n \longrightarrow \mathbf{2} \\ \mathsf{out}_{\langle i,j\rangle} &= \pi_{j+(n\times(i-1))} \end{aligned}$$

Now, the input to cell $\langle i, j\rangle$ *is simply the* split *of the outputs of its neighbours,* i.e.,

$$\begin{aligned} \mathsf{in}_{\langle i,j\rangle} &: \mathbf{2}^n \longrightarrow \mathbf{2}^4 \\ \mathsf{in}_{\langle i,j\rangle} &= \langle \mathsf{out}_{\langle i,\mathsf{dec}_n j\rangle}, \mathsf{out}_{\langle \mathsf{dec}_n i,j\rangle}, \mathsf{out}_{\langle i,\mathsf{inc}_n j\rangle}, \mathsf{out}_{\langle \mathsf{inc}_n i,j\rangle}\rangle \end{aligned}$$

where $\mathsf{dec}_n x = (x = 1 \rightarrow n, x-1)$ *and* $\mathsf{inc}_n x = (x = n \rightarrow 1, x+1)$. *Finally,* Bus *is defined as the lifting of the* split

$$\mathsf{w} \;=\; \langle \mathsf{in}_{\langle i,j \rangle} \mid i,j \in 1..n \rangle$$

The game of life *component is then written as*

$$\mathsf{GameLife} \;=\; ((\mathsf{Cell} \boxtimes \mathsf{Cell} \boxtimes \cdots \boxtimes \mathsf{Cell}) \,;\, \mathsf{Bus}) \,\urcorner$$

where

$$\mathsf{Bus} \;=\; \ulcorner \mathsf{w} \urcorner$$

Note how the hook *combinator is responsible for extending the game's behaviour to the infinite, once the component has been stimulated with an initial input.*

3 Behavioural Refinement

3.1 Component's Behaviour and Bisimulation

Successive observations of (or experiments with) a T-coalgebra p reveals its behavioural patterns. These are defined in terms of the results of the observers as recorded in the shape T. Then, just as the initial algebra is canonnically defined over the terms generated by successive application of constructors, it is also possible to define a canonical coalgebra in terms of such 'pure' observations. Such a coalgebra is the final object in $\mathsf{C_T}$, if it exists, and will be denoted by $\mathsf{out_T}$ over carrier ν_T.

Being *final* means that there exists a *unique* morphism to $\mathsf{out_T}$ from each other coalgebra $\langle U, p \rangle$. This is called the *coinductive extension* of p [48] or the *anamorphism* generated by p [28], and written as $[\!(p)\!]_\mathsf{T}$ or, simply, $[\!(p)\!]$, if the functor is clear from context. In other words, an anamorphism is defined as the unique function making the following diagram to commute:

$$\begin{array}{ccc} \nu_\mathsf{T} & \xrightarrow{\mathsf{out_T}} & \mathsf{T}\ \nu_\mathsf{T} \\ {\scriptstyle [\!(p)\!]_\mathsf{T}}\uparrow & & \uparrow{\scriptstyle \mathsf{T}\ [\!(p)\!]_\mathsf{T}} \\ U & \xrightarrow{p} & \mathsf{T}\ U \end{array}$$

or, alternatively, by the following universal law:

$$k = [\!(p)\!]_\mathsf{T} \;\;\Leftrightarrow\;\; \mathsf{out_T} \cdot k = \mathsf{T}\ k \cdot p \tag{10}$$

For each $u \in U$, $[\!(p)\!]_\mathsf{T}\ u$ can be thought of as the (observable) behaviour of a sequence of p transitions starting at state u. This explains yet another alternative

designation for an anamorphism: *unfold* [14]. On its turn, u in $[\![p]\!]_{\mathsf{T}}\ u$, is called the *seed* of the anamorphism.

As in the algebraic case, the *existence* part of the universal property (*i.e.*, the implication from left to right) provides a *definition* principle for (circular) functions to the final coalgebra which amounts to equip their source with a coalgebraic structure specifying the 'one-step' dynamics. Then the corresponding anamorphism gives the rest. In other words, such functions are defined by specifying their output under all different observers. The *uniqueness* part (*i.e.*, the reverse implication), on the other hand, offers a powerful *proof* principle — *coinduction*.

Agreeing with the intuition that the final coalgebra is the coalgebra of *all* behaviours, *observational equivalence* can be defined as

$$u \sim v \iff [\![p]\!]\ u = [\![q]\!]\ v \tag{11}$$

for u and v in the carriers of coalgebras $\langle U, p\rangle$ and $\langle V, q\rangle$, respectively. The notion of a *bisimulation*, which is central in coalgebra theory [42], entails a *local* proof theory for observational equivalence. Informally, two states of a T-coalgebra (or of two different T-coalgebras) are related by a bisimulation if their observation produces equal results and this is maintained along all possible transitions. I. e., each one can mimic the other's evolution. Originally the notion of bisimulation, which can be traced back to [44] and [12], was introduced in the context of process calculi in Park's landmark paper [38]. Later [2] gave a categorical definition which applies, not only to the kind of transition systems underlying the operational semantics of process calculi, but also to arbitrary coalgebras. Bisimulation *acquired a shape*: the shape of the chosen observation interface T.

A notion of *refinement* should also be shaped by T. Intuitively component p is a *behavioural refinement* of q if the behaviour patterns observed from p are a *structural restriction*, with respect to the *behavioural model* captured by monad B, of those of q. To make precise such a 'definition' we shall first describe behaviour patterns concretely as *generalized transitions*.

3.2 Refinement

Just as transition systems can be coded back as coalgebras, any coalgebra $\langle U, \alpha : \mathsf{T}U \longleftarrow U\rangle$ specifies a (T-shaped) transition structure over its carrier U. For extended polynomial Set endofunctors[6] such a structure may be expressed as a binary relation ${}_{\alpha}\longleftarrow : U \longleftarrow U$, defined in terms of the *structural membership* relation (which is an instance of generic datatype membership [19]) $\in_{\mathsf{T}} : U \longleftarrow \mathsf{T}\ U$, *i.e.*,

$$u'\ {}_{\alpha}\longleftarrow u \ \equiv\ u' \in_{\mathsf{T}} \alpha\ u$$

[6] The class inductively defined as the least collection of functors containing the identity Id and constant functors K for all objects K in the category, closed by functor composition and finite application of product, coproduct, covariant exponential and finite powerset functors.

or, in an equivalent but pointfree formulation which often simplifies formal reasoning, as the following relational equality[7]

$$_{\alpha}\longleftarrow \;=\; \in_{\mathsf{T}} \cdot \alpha$$

where $\in_{\mathsf{T}}$ is defined by induction on the structure of T:

$$\begin{array}{rcl}
x \in_{\mathsf{Id}} y & \text{iff} & x = y \\
x \in_{K} y & \text{iff} & \mathsf{false} \\
x \in_{\mathsf{T}_1 \times \mathsf{T}_2} y & \text{iff} & x \in_{\mathsf{T}_1} \pi_1\ y \ \vee\ x \in_{\mathsf{T}_2} \pi_2\ y \\
x \in_{\mathsf{T}_1 + \mathsf{T}_2} y & \text{iff} & \begin{cases} y = \iota_1\ y' & \Rightarrow x \in_{\mathsf{T}_1} y' \\ y = \iota_2\ y' & \Rightarrow x \in_{\mathsf{T}_2} y' \end{cases} \\
x \in_{\mathsf{T}^K} y & \text{iff} & \exists_{k \in K}.\ x \in_{\mathsf{T}} y\ k \\
x \in_{\mathcal{P}\mathsf{T}} y & \text{iff} & \exists_{y' \in y}.\ x \in_{\mathsf{T}} y'
\end{array}$$

For any function h, relation $\in_{\mathsf{T}}$ satisfies the following naturality condition

$$h \cdot \in_{\mathsf{T}} \;=\; \in_{\mathsf{T}} \cdot \mathsf{T}\ h \tag{12}$$

which can be proved by induction on T. Applying *shunting*[8] to the left to rigth inclusion component of equation (12) leads to

$$\in_{\mathsf{T}} \;\subseteq\; h^{\circ} \cdot \in_{\mathsf{T}} \cdot \mathsf{T}h \tag{13}$$

The dynamics of a component $p : O \longleftarrow I$ is based on functor $\mathsf{B}(\mathsf{Id} \times O)^I$. Therefore a possible (and intuitive) way of regarding component p as a behavioural refinement of q is to consider that p transitions are simply *preserved* in q. For non deterministic components this is understood as set inclusion. But one may also want to consider additional restrictions. For example, to stipulate that if p has no transitions from a given state, q should also have no transitions from the corresponding state(s). In any case the basic question is: how can such a refinement situation be identified?

The general 'recipe' to identify a refinement situation is to look for an *abstraction* to witness it [18]. In other words: look for a morphism in the relevant category, from the 'concrete' to the 'abstract' model such that the latter can be *recovered* from the former up to a suitable notion of equivalence, though, typically, not in a unique way. Component morphisms, however, are (seed preserving) coalgebra morphisms which are known to entail bisimilarity. Actually a T-coalgebra morphism $h : \beta \longleftarrow \alpha$ is a function from the state space of α to that of β such that

$$\mathsf{T}h \cdot \alpha \;=\; \beta \cdot h \tag{14}$$

[7] In the sequel both functional and relational composition will be denoted by the same symbol $\cdot$ given that the former is just a special case of the latter.

[8] In the relational calculus [4] Galois connection $f \cdot R \subseteq S \;\equiv\; R \subseteq f^{\circ} \cdot S$, involving function f and arbitrary relations R and S, is known as the *shunting rule*. Also note that notation R° stands for the *converse* of relation R.

Regarding α and β as (generalised) transition systems equation (14) becomes a relational equality:

$$h \cdot {}_{\alpha}\longleftarrow \;=\; {}_{\beta}\longleftarrow \cdot h \tag{15}$$

i.e., the conjunction of inclusions

$$h \cdot {}_{\alpha}\longleftarrow \;\subseteq\; {}_{\beta}\longleftarrow \cdot h \tag{16}$$

$${}_{\beta}\longleftarrow \cdot h \;\subseteq\; h \cdot {}_{\alpha}\longleftarrow \tag{17}$$

which, by introducing variables and observing that by *shunting* inclusion (16) can also be presented as ${}_{\alpha}\longleftarrow \subseteq h^{\circ} \cdot {}_{\beta}\longleftarrow \cdot h$, takes the following more familiar shape

$$u' \; {}_{\alpha}\longleftarrow u \;\Rightarrow\; h\, u' \; {}_{\beta}\longleftarrow h\, u \tag{18}$$

$$v' \; {}_{\beta}\longleftarrow h\, u \;\Rightarrow\; \exists_{u' \in U} .\; u' \; {}_{\alpha}\longleftarrow u \;\wedge\; u' = h\, v' \tag{19}$$

They jointly state that, not only α dynamics, as represented by the induced transition relation, is *preserved* by h (16), but also β dynamics is *reflected* back over the same h (17). Is it possible to weaken the morphism definition to capture only one of these aspects? In [29] this question got an afirmative answer, resorting to the notion of a preorder $\leq$ on a Set endofunctor T introduced in [21]. Such a preorder is defined as a functor $\leq$ in such a way that, for any function $h : V \longleftarrow U$, $\mathsf{T}h$ preserves the order, *i.e.*

$$x_1 \leq_{\mathsf{T}X} x_2 \;\Rightarrow\; (\mathsf{T}h)\, x_1 \leq_{\mathsf{T}Y} (\mathsf{T}h)\, x_2 \tag{20}$$

or, in a pointfree formulation,

$$(\mathsf{T}h) \cdot \leq \;\subseteq\; \leq \cdot (\mathsf{T}\, h) \tag{21}$$

Let us denote by $\dot{\leq}$ the pointwise lifting of $\leq$ to the functional level, *i.e.*

$$f \;\dot{\leq}\; g \;\equiv\; \forall_x .\; f\, x \leq g\, x \tag{22}$$

which can also be formulated in the following pointfree way as

$$f \;\dot{\leq}\; g \;\equiv\; f \subseteq \leq \cdot g \tag{23}$$

In [30] it is shown that, for any function h monotonic with respect to $\leq$ one has

$$f \;\dot{\leq}\; g \Rightarrow h \cdot f \;\dot{\leq}\; h \cdot g \tag{24}$$

$$f \;\dot{\leq}\; g \Rightarrow f \cdot h \;\dot{\leq}\; g \cdot h \tag{25}$$

In this context the main result in the above mentioned reference is the definition of a *forward* morphism $h : \beta \longleftarrow \alpha$ with respect to $\leq$ as a function from U to V such that

$$\mathsf{T}\, h \cdot \alpha \;\dot{\leq}\; \beta \cdot h$$

and the proof that

Lemma 1. *For* T *an endofunctor in* Set, T*-coalgebras and forward morphisms define a category. Moreover, forward morphisms preserve transitions if* $\leq$ *is compatible with the membership relation,* i.e.*, for all* $x \in X$ *and* $x_1, x_2 \in \mathsf{T}X$,

$$x \in_{\mathsf{T}} x_1 \ \wedge \ x_1 \leq \ x_2 \ \ \Rightarrow \ \ x \in_{\mathsf{T}} x_2 \tag{26}$$

or, in a pointfree formulation,

$$\in_{\mathsf{T}} \cdot \leq \ \subseteq \ \ \in_{\mathsf{T}} \tag{27}$$

Proof. We prove only the second part (see [30] for the full proof). Let h be a forward morphism. Transition preservation follows from

$$\begin{array}{ll}
 & {}_{\alpha}\longleftarrow \\
= & \quad \{ \text{ definition } \} \\
 & \in_{\mathsf{T}} \cdot \alpha \\
\subseteq & \quad \{ \text{ (13), monotonicity } \} \\
 & h^{\circ} \cdot \in_{\mathsf{T}} \cdot \mathsf{T}\, h \cdot \alpha \\
\subseteq & \quad \{ \text{ } h \text{ forward entails } \mathsf{T}h \cdot \alpha \ \subseteq \leq \cdot \beta \cdot h, \text{ monotonicity } \} \\
 & h^{\circ} \cdot \in_{\mathsf{T}} \cdot \leq \cdot \beta \cdot h \\
\subseteq & \quad \{ \text{ compatibility with } \in_{\mathsf{T}} \text{ (27), monotonicity } \} \\
 & h^{\circ} \cdot \in_{\mathsf{T}} \cdot \beta \cdot h \\
= & \quad \{ \text{ definition } \} \\
 & h^{\circ} \cdot \ {}_{\beta}\longleftarrow \cdot h
\end{array}$$

□

A preorder $\leq$ on an endofunctor T satisfying inclusion (27) will be referred to, in the sequel, as a *refinement preorder*. Then, the existence of a *forward* morphism connecting two components p and q witnesses a refinement situation whose symmetric closure coincides, as expected, with bisimulation. Formally,

Definition 7. *Component* p *is a behaviour refinement of* q*, written* $q \trianglelefteq p$*, if there exist components* r *and* s *and a (seed preserving) forward morphism* h *such that*

$$q \ \sim \ s \ \xleftarrow{\ h\ } \ r \ \sim \ p$$

The exact meaning of a refinement assertion $q \trianglelefteq p$ depends, of course, on the concrete refinement preorder $\leq$ adopted. But what do we know about such preorders? Condition (27) provides an upper bound leading to a definition of *structural inclusion*:

$$x \subseteq_{\mathsf{Id}} y \ \ \mathsf{iff} \ \ \forall_{e \in_{\mathsf{T}} x} .\ \ e \in_{\mathsf{T}} y \tag{28}$$

Several other cases arise by suitable restrictions. For example,

– Structural inclusion as defined above is too large to be useful in practice. Actually its definition on a constant functor is the universal relation which

would make refinement blind to the outputs produced. This suggests an additional requirement on refinement preorders for Cp components: their definition on a constant functor K must be equality on set K, *i.e.*, $x \leq_K y$ iff $x =_K y$ so that transitions with different O-labels could not be related. Note that refinement of non deterministic components based on this preorder captures the classical notion of *non determinism reduction*.

– A 'failure forcing' variant — $\subseteq^E_{\mathsf{T}}$, where E stands for 'emptyset' — guarantees that the concrete component fails no more than the abstract one. It is defined as $\subseteq_{\mathsf{T}}$ by replacing the clause for the powerset functor by

$$x \subseteq^E_{\mathcal{P}\mathsf{T}} y \quad \text{iff} \quad (x = \emptyset \Rightarrow y = \emptyset) \ \wedge \ \forall_{e \in x} \exists_{e' \in y}.\ e \subseteq_{\mathsf{T}} e'$$

– For partial components refinement based on the preorders above collapse into bisimilarity instead of entailing an increase of definition in the implementation side. An alternative is relation $\subseteq^F_{\mathsf{T}}$ (F standing for 'failure') which adds a *maybe* clause

$$x \subseteq^F_{\mathsf{T}+\mathbf{1}} y \quad \text{iff} \quad \begin{cases} x = \iota_1\ x' \wedge y = \iota_1\ y' & \Rightarrow x' \subseteq_{\mathsf{T}} y' \\ x = \iota_2\ * & \Rightarrow \mathsf{true} \end{cases}$$

taking precedence over the general sum clause.

We end this section with a small example. The reader is referred to [30] for a glimpse of a refinement calculus for state based components based on the existence of forward morphisms.

Example 3. *For an example of behavioural refinement consider a new specification of component* Store *which differs from the one in example 1 only in the specification of operation* pick*. The idea is that instead of choosing the message to be returned non deterministically from the set of messages waiting for more than a specified ϵ delay, the operation selects the message in that set which is waiting for a longer time. Formally, assuming function* lwait *computes such message, the specification becomes*

$$\begin{aligned} a_{\mathsf{Store}}\langle u, \mathsf{pick}\rangle \ = \ & \text{let } c = \{\langle m, t\rangle \in u \mid t - \mathsf{ttag}() \geq \epsilon\} \\ & \text{in } (c = \emptyset \ \rightarrow \ \emptyset, \text{ let } \langle m, t\rangle = \mathsf{lwait}\, c \text{ in } \{\langle u \setminus \{\langle m, t\rangle\}, \iota_2\ m\rangle\}) \end{aligned}$$

where notation $(\phi \ \rightarrow \ f, g)$ reads if ϕ then f else g*. Clearly, the identity morphism from this new coalgebra to the one in example 1 is a forward morphism witnessing the former as a refinement of the latter.*

4 Data Refinement

4.1 State Refinement

In the previous section component's refinement was discussed at the *behaviour* level based on a refinement preorder with respect to monad B. There is, however,

another axis for refinement: the *data* level, which amounts to the static refinement of the data structure which specifies the component state space. Data refinement, as discussed in formal development methodologies such as VDM [23], is the process of transforming abstract data structures into more concrete ones, a transformation which presumably entails efficiency (*e.g.*, the conversion of inductive data types into 'pointer'-based representations).

Refinement of a component specification p into another specification q has to fulfil a number of requirements. First of all, the existence of *enough redundancy* in the state space of q to represent all the elements in p is required. This is called in [23] the *adequacy* requirement and is captured by the definition of a surjection from the state space of q to that of p, called the *abstraction* or *retrieve* map. Next, *substitution* is regarded as 'complete' in the sense that (concrete) actions in q accept all the input values accepted by the corresponding abstract ones, and, for the same inputs, the results produced are also the same, up to the retrieve map. If components are specified, as they usually are in VDM, by pre and post-conditions, this amounts to say that, under refinement, neither pre-conditions are strengthened, nor post-conditions are weakened. Note this approach to data refinement, which can be traced back to Hoare's landmark paper [18], is usual in *model-oriented* design methods, even though several variants and alternatives have been proposed in the literature (see [41] for a recent account).

In this section we shall resort to SETS [34,35] — a calculus of data representations, based on identical principles: any refinement is witnessed by a surjection which, whenever partial, may induce a *representation invariant* on the concrete side. Each concrete operation is then *calculated* (rather than 'conjectured and verified' as in VDM) by solving the corresponding refinement diagram.

The calculus consists of inequations of the form $A \leq B$ (read: *data type B refines or implements data type A*) which witnesses the existence of an *abstraction* map abs from B to A with a *right-inverse* rep (called the *representation* relation), *i.e.*,

$$A \leq B \quad \text{iff} \quad A \underset{\mathsf{rep}}{\overset{\mathsf{abs}}{\leftarrow \leq \rightarrow}} B \quad \text{such that} \quad \mathsf{abs} \cdot \mathsf{rep} = \mathsf{id}_A \tag{29}$$

Note that rep is *injective* because $ker\,\mathsf{rep} \subseteq ker\,\mathsf{abs}^\circ$ and $ker\,\mathsf{abs}^\circ\, img\,\mathsf{abs}$ which coincides with identity as abs is surjective[9]. Moreover if abs is *partial*, the characteristic predicate of the codomain of relation rep defines the *invariant* induced by the refinement process.

Clearly (see [35] for a proof), the refinement relation is a preorder and is preserved by extended polynomial functors, *i.e.*,

$$A \underset{\mathsf{rep}}{\overset{\mathsf{abs}}{\leftarrow \leq \rightarrow}} B \quad \Rightarrow \quad \mathsf{T}\,A \underset{\mathsf{T\,rep}}{\overset{\mathsf{T\,abs}}{\leftarrow \leq \rightarrow}} \mathsf{T}\,B \tag{30}$$

[9] Notation $ker\,R$ (respectively, $img\,R$) stand for the kernel (respectively, image) of relation R defined as $ker\,R = R^\circ \cdot R$ (respectively, $img\,R = R \cdot R^\circ$) [4].

Example 4. *A simple example of data refinement in the context of the* Store *component is the implementation of its state space* $U = \mathcal{P}(M \times T)$ *as a finite sequence of type* $(M \times T)^*$ *with* $\mathsf{abs} = \mathsf{elems}$*, the function which returns the set of elements of a list. Other representations for sets, including the notion of a* bag *(which retains the unordered structure of set while keeping track of element repetition) are recorded in the following inequations:*

$$\mathcal{P}A \;\leq\; Nat \leftharpoonup A \;\leq\; A^* \tag{31}$$

where notation $B \leftharpoonup A$ *stands for a partial function (also called a simple relation in [4] or a* mapping *in specification methods like* VDM *[23]) from* A *to* B*. An elementary example of a data refinement situation where the abstraction morphism is not a function is the following representations of elements as 'pointers':*

$$A \underset{\mathsf{rep}=\iota_1}{\overset{\mathsf{abs}=\iota_1^\circ}{\leq}} A + \mathbf{1} \tag{32}$$

which, moreover, induce the concrete invariant $\phi = [\mathsf{true}, \mathsf{false}]$ *over the implementation type. References [34,35] and [36] provide several applications of this calculus to the derivation of imperative programs and data base schemes.*

Once the state space of a component $p = \langle u \in U, \alpha : \mathsf{T}U \longleftarrow U \rangle$ is refined into, say, V a new component is defined over V, whose seed is given by $\mathsf{abs}\, u$ and the dynamics $\beta : \mathsf{T}V \longleftarrow V$ computed as a solution to the following equation

$$\alpha \;=\; \mathsf{T}\mathsf{abs} \cdot \beta \cdot \mathsf{rep} \tag{33}$$

The basic result, from the point of view of a component calculus, is that data refinement entails bisimilarity, *i.e.*,

Lemma 2. *Components* p *and* q *as defined above satisfy* $p \sim q$*.*

Proof. Consider the general case in which refinement $U \leq V$, witnessed by abs and rep, induces a concrete invariant ϕ over V, *i.e.*, an inclusion $i : V \longleftarrow V_\phi$. Let also β' denote the restriction of β to V_ϕ. The starting point is equation (33) which defines the dynamics of q. The target is to show that $\mathsf{abs}_\phi = \mathsf{abs} \cdot i$ is a coalgebra morphism. I.e.,

$$\begin{array}{ll} & \alpha \cdot \mathsf{abs}_\phi \;=\; \mathsf{T}\mathsf{abs}_\phi \cdot \beta' \\ \equiv & \quad \{\; \mathsf{abs}_\phi = \mathsf{abs} \cdot i \;\} \\ & \alpha \cdot \mathsf{abs} \cdot i \;=\; \mathsf{T}\mathsf{abs} \cdot \mathsf{T}i \cdot \beta' \\ \equiv & \quad \{\; i \text{ is a coalgebra morphism, } \textit{i.e.}, \; \beta \cdot i = \mathsf{T}i \cdot \beta' \;\} \\ & \alpha \cdot \mathsf{abs} \cdot i \;=\; \mathsf{T}\mathsf{abs} \cdot \beta \cdot i \\ \Rightarrow & \quad \{\; i \text{ is injective} \;\} \\ & \alpha \cdot \mathsf{abs} \;=\; \mathsf{T}\mathsf{abs} \cdot \beta \end{array}$$

$$
\begin{array}{ll}
\Rightarrow & \{ \text{ Leibniz equality } \} \\
& \alpha \cdot \mathsf{abs} \cdot \mathsf{rep} \;=\; \mathsf{T}\mathsf{abs} \cdot \beta \cdot \mathsf{rep} \\
\equiv & \{ \text{ } \mathsf{abs} \cdot \mathsf{rep} = \mathsf{id} \text{ and } (33) \} \\
& \alpha \;=\; \alpha
\end{array}
$$

4.2 Shape Refinement

In the approach to component modelling discussed in this paper, *interfaces* are encoded in the shape of functor T corresponding to the component's service signature. Therefore applying data refinement to this level may capture some forms of interface *enrichment*. Consider, for example, the elementary cases of adding an attribute or an operation to the *shape* of a (deterministic) component:

– adding an *attribute* $\mathsf{at} : B \longleftarrow X$

$$A \times X \underset{}{\overset{\mathsf{abs}=\pi_1 \times \mathsf{id}}{\leftarrow\!\!\rightarrow}} (A \times B) \times X$$

– adding an *operation* $\mathsf{op} : X \longleftarrow X \times B$

$$X^A \underset{}{\overset{\mathsf{abs}=(\cdot \iota_1)}{\leftarrow\!\!\rightarrow}} X^{A+B}$$

Now the interesting result is that refinement of the signature functor has a counterpart at the behavioural level, *i.e.*, the carriers of the corresponding final coalgebras, which form the spaces of their behaviours, are also related by a data refinement. Formally, we prove that the data refinement relation as introduced above extends to coinductive types:

Lemma 3. *Let* T *and* G *be extended polynomial functors. Then,*

$$\mathsf{T}X \underset{\mathsf{rep}}{\overset{\mathsf{abs}}{\leftarrow\!\!\leq\!\!\rightarrow}} \mathsf{G}X \quad \Rightarrow \quad \nu_{\mathsf{T}} \underset{\mathsf{rep}_\nu}{\overset{\mathsf{abs}_\nu}{\leftarrow\!\!\leq\!\!\rightarrow}} \nu_{\mathsf{G}} \tag{34}$$

where ν_{T} *denotes the carrier of the final* T*-coalgebra. Moreover,*

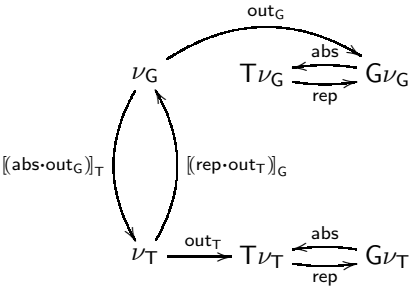
$$\mathsf{abs}_\nu \;\triangleq\; [\![\mathsf{abs} \cdot \mathsf{out}_{\mathsf{G}}]\!]_{\mathsf{T}} \quad \textit{and} \quad \mathsf{rep}_\nu \;\triangleq\; [\![\mathsf{rep} \cdot \mathsf{out}_{\mathsf{T}}]\!]_{\mathsf{G}}$$

for abs *and* rep natural *on* X

Proof. We have to show that

$$[\![\mathsf{abs} \cdot \mathsf{out}_{\mathsf{G}}]\!]_{\mathsf{T}} \;\cdot\; [\![\mathsf{rep} \cdot \mathsf{out}_{\mathsf{T}}]\!]_{\mathsf{G}} \;=\; \mathsf{id} \tag{35}$$

in the context of the following diagram:



Therefore[10],

$$
\begin{array}{ll}
& [\![\mathsf{abs}\cdot\mathsf{out_G}]\!]_\mathsf{T}\ \cdot\ [\![\mathsf{rep}\cdot\mathsf{out_T}]\!]_\mathsf{G}\ =\ \mathsf{id} \\
\equiv & \quad \{\ \text{re'ection for coinductive extension}\ \} \\
& [\![\mathsf{abs}\cdot\mathsf{out_G}]\!]_\mathsf{T}\ \cdot\ [\![\mathsf{rep}\cdot\mathsf{out_T}]\!]_\mathsf{G}\ =\ [\![\mathsf{out_T}]\!]_\mathsf{T} \\
\Leftarrow & \quad \{\ \text{fusion for coinductive extension}\ \} \\
& \mathsf{abs}\cdot\mathsf{out_G}\cdot[\![\mathsf{rep}\cdot\mathsf{out_T}]\!]_\mathsf{G}\ =\ \mathsf{T}[\![\mathsf{rep}\cdot\mathsf{out_T}]\!]_\mathsf{G}\cdot\mathsf{out_T} \\
\equiv & \quad \{\ \text{cancellation for coinductive extension}\ \} \\
& \mathsf{abs}\cdot\mathsf{G}[\![\mathsf{rep}\cdot\mathsf{out_T}]\!]_\mathsf{G}\cdot\mathsf{rep}\cdot\mathsf{out_T}\ =\ \mathsf{T}[\![\mathsf{rep}\cdot\mathsf{out_T}]\!]_\mathsf{G}\cdot\mathsf{out_T} \\
\equiv & \quad \{\ \mathsf{rep}\ \text{is natural}\} \\
& \mathsf{abs}\cdot\mathsf{rep}\cdot\mathsf{T}[\![\mathsf{rep}\cdot\mathsf{out_T}]\!]_\mathsf{G}\cdot\mathsf{out_T}\ =\ \mathsf{T}[\![\mathsf{rep}\cdot\mathsf{out_T}]\!]_\mathsf{G}\cdot\mathsf{out_T} \\
\equiv & \quad \{\ \text{hip}\ \} \\
& \mathsf{T}[\![\mathsf{rep}\cdot\mathsf{out_T}]\!]_\mathsf{G}\cdot\mathsf{out_T}\ =\ \mathsf{T}[\![\mathsf{rep}\cdot\mathsf{out_T}]\!]_\mathsf{G}\cdot\mathsf{out_T}
\end{array}
$$

Example 5. *Another typical example is what could be called* stream completion *induced by the following data refinement at the signature level:*

$$\mathbf{1} + A \times X \overset{\mathsf{abs}=!+\mathsf{id}}{\underset{}{\longleftarrow}} (A + \mathbf{1}) \times X \cong X + A \times X$$

Note that the final coalgebra for the 'abstract' shape is A^∞, i.e., *the space of finite and infiinite sequences of* A, *whereas for the concrete case one gets* $(A+\mathbf{1})^\omega$, i.e.,

[10] The laws of *reflection*, *cancellation* and *fusion* stated below, in this order, and used in the proof are standard results on coinduction easily derived from the universal property (10) [15].

$$
\begin{array}{rcl}
[\![\mathsf{out_T}]\!] & = & \mathsf{id}_{\nu_\mathsf{T}} \\
\mathsf{out_T}\cdot[\![p]\!] & = & \mathsf{T}\,[\![p]\!]\cdot p \\
[\![p]\!]\cdot h & = & [\![q]\!] \quad \text{if} \quad p\cdot h\ =\ \mathsf{T}\,h\cdot q
\end{array}
$$

streams of either elements of A or a mark $* \in \mathbf{1}$. By the lemma above one may easily conclude that $A^\infty \leq (A+\mathbf{1})^\omega$, a fact often used in coalgebraic specification [22], where a finite sequence is extended to a stream by replication of a dummy value.

5 Conclusions and Further Work

This paper provided an overview of an approach to refinement of (state-based) components whose main theory has been developed in previous publications (namely, [29,30]). The integration of behaviour and data refinement and the application of the latter to a form of interface enrichement is, however, new.

The main possible interest of this approach is its parametrization by a model of behaviour captured by a strong monad B. This is *generic* enough to capture a number of situations, depending on both B and the refinement preorder adopted. Non determinism reduction is one possibility among many others. For example, Poll's notion of *behavioural subtyping* in [39], at the model level, also emerges as another instantiation.

A note on related work is now in order. First of all two major influences should be acknowledged. The first one relates to the use of a bicategorical setting to capture the 'two-level structure' in component models which is in debt to previous work by R. Walters and his collaborators on models for deterministic input-driven systems [24,25]. The other is the recent area of coalgebraic specification of object-oriented systems (see *e.g.*, [40,20]), which has been developed with a similar motivation, although in a property-oriented, or axiomatic, framework.

An alternative, but related, approach to componentware is inspired by research on coordination languages [17,37] and favors strict component decoupling in order to support a looser inter-component dependency. Here computation and coordination are clearly separated, communication becomes *anonymous* and component interconnection is externally controlled. This model is (partially) implemented in JavaSpaces on top of Jini [33] and fundamental to a number of approaches to componentware which identify communication by generic channels as the basic interaction mechanism — see, *e.g.*, Reo [3], Piccola [43,32], as well as [16,13] or [10].

The genericity of the approach described in this paper and its coalgebraic basis seems promising, although a lot of work remains to be done. Among the current research directions we would like to underline the following two.

Backwards refinement. Behavioural refinement was defined in section 3 in terms of transition preservation, *i.e.*, as a sort of T-shaped simulation witnessed by what we have called *forward* morphisms. An alternative point of view is based on the dual notion of *backward* morphisms, morphisms which verify

$$\beta \cdot h \;\dot{\leq}\; \mathsf{T}\, h \cdot \alpha \tag{36}$$

In [29,30] these are shown to form a category and to *reflect* transitions, in the sense of equation (19), although possible applications to component refinement are still to be developed.

Induced Distribution. Several laws of data refinement split components' state space into a number of factors. Typical examples are laws whose purpose is to factorize mappings with structured domains or codomains, heavily used in the derivation of database implementations [36]. For example the following laws, studied in [35], refine a mapping to either a sum or a product type into a product of two mappings:

$$(B + C) \leftharpoonup A \quad \underset{\text{cojoin}}{\leq} \quad (B \leftharpoonup A) \times (C \leftharpoonup A)$$

$$(B \times C) \leftharpoonup A \quad \underset{\text{join}}{\leq} \quad (B \leftharpoonup A) \times (C \leftharpoonup A)$$

where abstractions are defined as

$$\begin{aligned} \mathsf{cojoin} &= \cup \cdot ((\iota_1 \cdot) \times (\iota_2 \cdot)) \\ \mathsf{join} &= \langle\ ,\ \rangle \end{aligned}$$

where $\langle R, S\rangle = \cap \cdot ((\pi_1^\circ \cdot R) \times (\pi_2^\circ \cdot S))$, is a relational *split* [15]. Similarly relational *either* witnesses the decomposition of a mapping from a sum type:

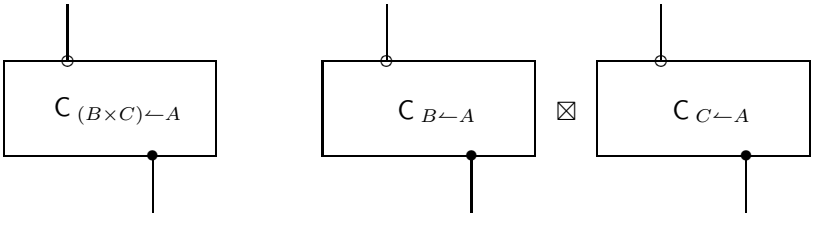
$$A \leftharpoonup (B + C) \quad \underset{\text{peither}}{\leq} \quad (A \leftharpoonup B) \times (A \leftharpoonup C)$$

with

$$\mathsf{peither} = [\ ,\ ]$$

where $[R, S] = (R \cdot \iota_1^\circ) \cup (S \cdot \iota_2^\circ)$.

The state space factorization underlying this sort of laws may lead to a similar component factorization by aggregation of the original actions according to the part of the state space they manipulate. This finds application in typical re-engineering processes in which clusters of related operations identified in monolythic code are coupled together around specific state loci. The process is suggested in the following diagram where data refinement induces the factorirization of the original component into two new ones which are composed in parallel.



Our main current research concern is the study of a precise characterization of this phenomonom. In particular, a suitable approach entails the need for *rethinking interfaces* in terms of decomposition of operations' signatures into pairs of input/output *ports* (as in *e.g.*, [10]) providing a basis for the specification of *component usage* as a transition structure over port names. In this context, representation *invariants* induced by data refinement (notice that both join and cojoin abstractions are partial) would generate aditional constraints over such *component usage* specifications.

Acknowledgements

This research was funded by the Portuguese Foundation for Science and Technology, in the context of the PURe project, under contract POSI/ICHS/44304/2002. The on-going collaboration with José Oliveira on the refinement problem is gratefully acknowledged.

References

1. J. R. Abrial. *The* B *Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. P. Aczel and N. Mendler. A final coalgebra theorem. In D. Pitt, D. Rydeheard, P. Dybjer, A. Pitts, and A. Poigne, editors, *Proc. Category Theory and Computer Science*, pages 357–365. Springer Lect. Notes Comp. Sci. (389), 1988.
3. F. Arbab. Abstract behaviour types: a foundation model for components and their composition. In F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, pages 33–70. Springer Lect. Notes Comp. Sci. (2852), 2003.
4. R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In B. Möller, H. Partsch, and S. Schuman, editors, *Formal Program Development*, pages 7–42. Springer Lect. Notes Comp. Sci. (755), 1993.
5. R. C. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In S. D. Swierstra, P. R. Henriques, and J. N. Oliveira, editors, *Third International Summer School on Advanced Functional Programming, Braga*, pages 28–115. Springer Lect. Notes Comp. Sci. (1608), September 1998.
6. L. S. Barbosa. Components as processes: An exercise in coalgebraic modeling. In S. F. Smith and C. L. Talcott, editors, *FMOODS'2000 - Formal Methods for Open Object-Oriented Distributed Systems*, pages 397–417. Kluwer Academic Publishers, September 2000.
7. L. S. Barbosa. Towards a Calculus of State-based Software Components. *Journal of Universal Computer Science*, 9(8):891–909, August 2003.
8. L. S. Barbosa and J. N. Oliveira. State-based components made generic. In H. Peter Gumm, editor, *CMCS'03, Elect. Notes in Theor. Comp. Sci.*, volume 82.1. Elsevier, 2003.
9. L. S. Barbosa, M. Sun, B. K. Aichernig, and N. Rodrigues. On the semantics of componentware: a coalgebraic perspective. In Jifeng He and Zhiming Liu, editors, *Mathematical Frameworks for Component Software: Models for Analysis and Synthesis*, Series on Component-Based Development. World Scientific, 2005.

10. M. A. Barbosa and L. S. Barbosa. Specifying software connectors. In K. Araki and Z. Liu, editors, *1st International Colloquium on Theorectical Aspects of Computing (ICTAC'04)*, pages 53–68, Guiyang, China, September 2004. Springer Lect. Notes Comp. Sci. (3407).
11. J. Benabou. Introduction to bicategories. *Springer Lect. Notes Maths. (47)*, pages 1–77, 1967.
12. J. van Benthem. *Modal Correspondence Theory.* Ph.D. thesis, University of Amsterdam, 1976.
13. K. Bergner, A. Rausch, M. Sihling, A. Vilbig, and M. Broy. A Formal Model for Componentware. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 189–210. Cambridge University Press, 2000.
14. R. Bird. *Functional Programming Using Haskell.* Series in Computer Science. Prentice-Hall International, 1998.
15. R. Bird and O. Moor. *The Algebra of Programming.* Series in Computer Science. Prentice-Hall International, 1997.
16. M. Broy. Semantics of finite and infinite networks of communicating agents. *Distributed Computing*, (2), 1987.
17. D. Gelernter and N. Carrier. Coordination languages and their significance. *Communication of the ACM*, 2(35):97–107, February 1992.
18. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.
19. P. F. Hoogendijk. *A generic theory of datatypes.* Ph.D. thesis, Department of Computing Science, Eindhoven University of Technology, 1996.
20. B. Jacobs. Objects and classes, co-algebraically. In C. Lengauer B. Freitag, C.B. Jones and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Academic Publishers, 1996.
21. B. Jacobs and J. Hughes. Simulations in coalgebra. In H. Peter Gumm, editor, *CMCS'03, Elect. Notes in Theor. Comp. Sci.*, volume 82.1, Warsaw, April 2003.
22. Bart Jacobs. Exercises in coalgebraic specification. In R. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 237–280. Springer Lect. Notes Comp. Sci. (2297), 2002.
23. Cliff B. Jones. *Systematic Software Development Using* VDM. Series in Computer Science. Prentice-Hall International, 1986.
24. P. Katis, N. Sabadini, and R. F. C. Walters. Bicategories of processes. *Journal of Pure and Applied Algebra*, 115(2):141–178, 1997.
25. P. Katis, N. Sabadini, and R. F. C. Walters. On the algebra of systems with feedback and boundary. *Rendiconti del Circolo Matematico di Palermo*, II(63):123–156, 2000.
26. A. Kock. Strong functors and monoidal monads. *Archiv für Mathematik*, 23:113–120, 1972.
27. G. H. Mealy. A method for synthesizing sequential circuits. *Bell Systems Techn. Jour.*, 34(5):1045–1079, 1955.
28. E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 1991 ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer Lect. Notes Comp. Sci. (523), 1991.
29. Sun Meng and L. S. Barbosa. On refinement of generic software components. In C. Rettray, S. Maharaj, and C. Shankland, editors, *10th Int. Conf. Algebraic Methods and Software Technology (AMAST)*, pages 506–520, Stirling, 2004. Springer Lect. Notes Comp. Sci. (3116). Best Student Co-authored paper Award.

30. Sun Meng and L. S. Barbosa. Components as coalgebras: The refinement dimension. *Theor. Comp. Sci. (accepted for publication)*, 2005.
31. R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall International, 1989.
32. O. Nierstrasz and F. Achermann. A calculus for modeling software components. In F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, pages 339–360. Springer Lect. Notes Comp. Sci. (2852), 2003.
33. S. Oaks and H. Wong. *Jini in a Nutshell*. O'Reilly and Associates, 2000.
34. J. N. Oliveira. A reification calculus for model-oriented software specification. *Formal Aspects of Computing*, 2(1):1–23, 1990.
35. J. N. Oliveira. Software reification using the SETS calculus. In *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. Springer-Verlag, 8–10 January 1992. (Invited paper).
36. J. N. Oliveira and C. J. Rodrigues. Transposing relations: From *Maybe* functions to hash tables. In D. Kozen, editor, *7th International Conference on Mathematics of Program Construction*, pages 334–356. Springer Lect. Notes Comp. Sci. (3125), July 2004.
37. G. Papadopoulos and F. Arbab. Coordination models and languages. In *Advances in Computers — The Engineering of Large Systems*, volume 46, pages 329–400. 1998.
38. D. Park. Concurrency and automata on infinite sequences. pages 561–572. Springer Lect. Notes Comp. Sci. (104), 1981.
39. Erik Poll. A coalgebraic semantics of subtyping. *Theorectical Informatica and Apllications*, 35(1):61–82, 2001.
40. H. Reichel. An approach to object semantics based on terminal co-algebras. *Math. Struct. in Comp. Sci.*, 5:129–152, 1995.
41. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
42. J. Rutten. Universal coalgebra: A theory of systems. *Theor. Comp. Sci.*, 249(1):3–80, 2000. (Revised version of CWI Techn. Rep. CS-R9652, 1996).
43. J.-G. Schneider and O. Nierstrasz. Components, scripts, glue. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures - Advances and Applications*, pages 13–25. Springer-Verlag, 1999.
44. K. Segerberg. An essay in classical modal logic. *Filosofiska Studier*, (13), 1971.
45. J. M. Spivey. *The Z Notation: A Reference Manual (2nd ed)*. Series in Computer Science. Prentice-Hall International, 1992.
46. C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
47. The RAISE Language Group. *The RAISE Specification Language*. Prentice Hall International, 1992.
48. D. Turi and J. Rutten. On the foundations of final coalgebra semantics: non-well-founded sets, partial orders, metric spaces. *Math. Struct. in Comp. Sci.*, 8(5):481–540, 1998.
49. P. Wadler and K. Weihe. Component-based programming under different paradigms. Technical report, Dagstuhl Seminar 99081, February 1999.

A Fully Abstract Semantics for UML Components

F.S. de Boer[1,2], M.M. Bonsangue[2,⋆], M. Steffen[3], and E. Ábrahám[3]

[1] CWI, Amsterdam, The Netherlands
frb@cwi.nl
[2] LIACS, Leiden University, The Netherlands
marcello@liacs.nl
[3] Christian-Albrechts-University, Kiel, Germany
{ms, eab}@informatik.uni-kiel.de

Abstract. We present a fully abstract semantics for components. This semantics is formalized in terms of a notion of trace for components, providing a description of the component externally observable behavior inspired by UML sequence diagrams. Such a description abstracts from the actual implementation given by UML state-machines. Our full abstraction result is based on a may testing semantics which involves a composition of components in terms of cross-border dynamic class instantiation through component interfaces.

1 Introduction

The Unified Modelling Language (UML)[18] is widely adopted as the de facto industry standard for modelling object-oriented software systems. It consists of several graphical notations providing different views of the system being modelled. There are two basic types of diagrams: behavior diagrams and structure diagrams. These diagrams include sequence diagrams, state machines, class diagrams and component diagrams.

We use UML for investigating features such as state encapsulation, and name-passing in synchronous communication in combination with dynamic class instantiation. Basically, in UML a component is a set of classes with explicit contextual dependencies. Some instances of classes of a component are called ports. Components can communicate only through their ports. Most importantly, a port of a component can also instantiate new ports of another component. The explicit context dependencies of a component guarantee that ports have enough structural information about the environment. However the behavior of such an external environment is not under control of the component itself. In other words, a component is an open program, with implementation code containing calls to operations and constructors of interfaces that are not bound to any particular behavior specification.

From the point of view of a component, the ports of other components belong to the environment, and are internally known only as typed identifiers. Although the behavior of the environment is not fixed at priori, it has to obey to certain laws. For example, because the state of a port is encapsulated, external ports *cannot* always communicate

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

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 49–69, 2005.
© Springer-Verlag Berlin Heidelberg 2005

with each other. To illustrate this, consider a port of a component i that creates two new ports e_1 and e_2 of some component in the environment. The ports e_1 and e_2 are both external, but unable to communicate with each other unless the internal object i let one of them know the identity of the other. The above situation is characteristic of a framework with dynamic scope: new clusters of objects that know each other can be created as new external instances appear, and old clusters may merge as a consequence of a communication.

1.1 Contribution of This Paper

In this paper we select a subset of UML notations suitable as basis for modelling component-based systems. Inspired by UML sequence diagrams, we give a denotational semantics to UML components in terms of traces of their externally observable events. A trace describes a sequence of interactions between the ports of a set of components. Here a port is an instance of a class of a component realizing one of its interface, and an interaction is a synchronization on an operation declared on one of the interface of a component.

We define an observational equivalence for components based on may testing, and show that ordinary traces are, in general, not fully abstract: two components can be observationally equivalent but their associated set of traces be different. Our main result is the characterization of trace abstractions that takes into account the clustering structure of objects dictated by their dynamic scope. These traces are full abstract with respect to may testing observational equivalence.

1.2 Related Work

There is an increasing interest to give a rigorous foundation to UML for addressing, e.g., the needs for modelling safety critical applications. Some approaches are based on translating UML subsets into existing formalisms, like the π-calculus [19], other have proposed new meta-modelling language calculi as foundation for the semantics of UML, e.g. [11]. In this paper we present a variant of the UML subset considered by Damm et al. and formalized as a transition system [12]. The most significant departures from this work are that we do not consider asynchronous inter-object communications and do not distinguish among active, reactive and passive objects.

There are several full abstraction results for may-testing semantics for calculi of processes interacting in dynamically changing communication topology [6,14]. The UML description of classes by state-machines combines mechanisms for dynamic process creation similarly to object calculi [1,10,20,16] with synchronization mechanisms as in process calculi [9,6,14].

The closest work to our is Jeffrey and Rathke [16] fully abstract semantics of concurrent objects. While our components are open, programs in [16] are closed, in the sense we explained above, since their creation of a new object involves the specification of the behavior of the newly created object. Consequently, in their setting, the environment can be basically viewed as a static and a priori given group of objects. This contrasts with our setting, where the program itself creates dynamically its own environment and imposes constraints on the communication topology of its environment.

Different from previous full abstraction results, the construction of a distinguishing context in the full abstraction proof requires a novel technique for the definition of a generic behavior capturing all instances of an external class. This we consider as one of the main technical contribution of our paper, that helps in a better understanding of the role of static class variables in class-based object-oriented languages like Java.

2 UML Classes, State-Machines and Components

Next we describe the subset of UML we use in this paper. We use UML as an inspiration source, and have no pretence of fully formalizing the numerous concepts used in the UML diagrams. UML is an object-oriented modelling technique based on the concept of class. A *class* is a named description of a set of objects. Its signature consists of a finite set of attributes and a finite set of operations (one of them declared as constructor). Attributes and operations are typed either by basic types (like integers and Boolean) or by the identifier of a class or of an interface. An *interface* is a named description of a set of operations. Differently from a class signature, an interface does not declare any attribute. We say that an interface is *realized* by a class (that for simplicity we assume carrying the same name) if the set of operations of the interface is included in that of the class realizing it.

An *object* is an instance of a class. There are different kinds of inter-object communications in UML. We consider only communication via *synchronous operations* restricting to operations with two parameter only: one for passing the identity of the caller of the operation, and another for passing a value (that we will often assume to be the identity of another object). The execution of a synchronous operation involves a synchronization on the execution of an operation call by the sender and a corresponding trigger by the receiver. Such a synchronization results in an *assignment* of the value of the actual parameters of the operation call to the instance variables of the receiver that appear as formal parameters of the operation.

In contrast to a synchronous operation, a *primitive operation* is an operation acting directly on the instance variables of the objects, without any synchronization. Therefore the meaning of a primitive operation is defined in terms of a state transformation.

2.1 Abstract State-Machines

In UML the behavior of an object is describe generically by means of an abstract state-machine associated to the class of which it is an instance. A *state-machine* is a kind of structured transition system that records the dependencies between the states of an object and its reaction to messages. More formally, a state machine associated to a class c consists of transitions of the form

$$l_1 \xrightarrow{[g]t/a}_c l_2$$

where l_1 is the entry location and l_2 is the exit location of the transition. Transitions may be *guarded* by a boolean guard g and labelled by a *trigger* t and an *action* a. The evaluation of the boolean guard g is assumed to be side-effect free.

A trigger t is of the form

$$op(x, y)$$

where op is the name of an operation (possibly the constructor) declared in the class c, while x, y are attributes of c used to store the identity of the caller and the value it pass when calling the operation.

An action a is either a primitive operation, a constructor call or a synchronous operation call. A constructor call is of the form $c.new(self, x)$, where new is the constructor of the class (or interface) c, and the attribute $self$ store the identity of t the caller object. The attribute x is typed by c and it will store the identity of the newly created object. A synchronous operation call is of the form

$$x.op(self, y)$$

where op is an operation declared in the class (or interface) typing the attribute x, that stores the identity of the callee of the operation. The attribute y is also declared in c and stores the value to be passed to the callee. We have not considered the more usual synchronous operations that return by means of a rendez-vous mechanism because we can encode this mechanism by means of an appropriate operation call and a respective trigger.

2.2 Components

In this paper we consider a *component* $\mathcal{C}$ as a part of a system consisting of a set of classes B and a set of interfaces $I = P \cup R$. Each class in B is associated with state-machine. The operations of the interfaces in I are typed only by other interfaces in the same set I. Interfaces in I can be either provided or required. Each *provided interface* $p \in P$ is realized by a class in B, and hence with the same name of p. A *required interface* $r \in R$ is an interface with a name different from that of any other class in B. It can be used by classes in B for typing their attributes. This way a component declares its dependencies on another components with interfaces in R as provided interfaces.

A class realizing a provided interface or depending on one or more required interfaces is called a *role*, and its instances are called *ports* [5]. An *internal class* is a class of a component that is not a role. Attributes of an internal class are typed only by primitive types or by classes within the same component, whereas attributes of a role may be typed also by the required interfaces. This means that a component is an open system, with its ports as the only points of interactions with environment: ports may be triggered by other ports in the environment, and call operations declared in the required interfaces, including the declared constructors. However, a class realizing a required interface is external, i.e., it belong to a different component. Encapsulation of the component internal implementation is ensured because instances of internal classes may synchronize only on operations of other objects within the same component, thus preventing a tight coupling between the component internal structure and the component environment.

Components can be composed by connecting the required interfaces of a constituent component with the provided interfaces (that for simplicity we assume to have the same name) that belongs to other constituent components. For simplicity we define interface

connection as set inclusion of operations. More formally, let $\mathcal{C}_1 = \langle B_1, P_1 \cup R_1 \rangle$ and $\mathcal{C}_2 = \langle B_2, P_2 \cup R_2 \rangle$ be two components. Their *composition* $\mathcal{C}_1 \oplus \mathcal{C}_2$ is defined as the component $\mathcal{C} = \langle B, I \rangle$ with $B = B_1 \cup B_2$ (that are assumed to be disjoint) and with $I = P \cup R$ obtained by taking $P = P_1 \cup P_2$ and $R = (R_1 \setminus P_2) \cup (R_2 \setminus P_1)$. For example, if one component provides all interfaces required by another one, then the component resulting from their composition has no required interfaces, and remains open to the environment only via its provided interfaces.

The above notion of component is inspired by that of UML as introduced in [18], but it differs in a number of crucial points. In particular, for simplicity we do not allow hierarchical composition of components (and hence we do not need delegation connectors), and, contrary to UML 2.0 we do not consider components as unit of instantiation but rather we consider a component as a static unit of abstraction with a dynamically growing number of ports.

2.3 Operational Semantics

Next we define the operational semantics of a component in terms of the abstract state machines associated with each of its constituent classes.

Let $Class$ be a set of class (and interface) identifiers, with typical element c, and assume given, for each class name c, an infinite set $Obj(c)$ of names for the instances of the class c. We denote by Obj the union of $Obj(c)$ for all $c \in Class$. Further, let Att be a set of attributes (including loc and $self$) and Val be a set of values (including the undefined value nil).

A *object diagram* σ of a component $\mathcal{C} = \langle B, I \rangle$ is a partial function in $Obj \rightharpoonup (Att \rightharpoonup Val)$ assigning values to attributes of the existing instances of classes in B. The domain of an object diagram σ is denoted by $dom(\sigma)$, and the value $\sigma(o)(x)$ of the instance variable x of the object o is denoted by $\sigma(o.x)$. For all $o \in dom(\sigma)$ we require that $\sigma(o.self) = o$ and that $o \in Obj(c)$ for some class c in B.

Control information of each object o in an object-diagram is given by $\sigma(o.loc)$, assuming for each class that the attribute loc is used only to refer to the current location of the state machine of the class of which o is an instance. An object diagram is called *initial* if the only attributes different from nil are $self$ and loc.

The operational semantics of a component $\mathcal{C} = \langle B, P \cup R \rangle$ is defined in terms of a *transition relation* $\longrightarrow$ between object diagrams labelled by externally observable *communication events* of the form

$$e.op(i, v) \quad \text{and} \quad i.op(e, v)\,, \tag{1}$$

where $e \in Obj(r)$, for some required interface $r \in R$, is the identity of an *external port*, and $i \in Obj(p)$, for some provided interface $p \in P$, is the identity of an *internal port* of $\mathcal{C}$. The idea is that i is an instance of the class of $\mathcal{C}$ realizing the interface p, whereas e is an instance of the class r realizing the interface r in another component. We will use this convention throughout this paper. The event $e.op(i, v)$ denotes the synchronization of the port e with the port i on the operation op provided by e. Similarly, $i.op(e, v)$ denotes the synchronization of the port i with the port e on an operation op provided by i. In both cases the synchronization involves the transmission of the value v.

We label the *transition relation* $\longrightarrow$ also with *creation events* of the form

$$new(o, u)$$

indicating the synchronization on the constructor new of the class c between the object creator o and the new instance u of c. As usual, a transition labelled by τ denotes an internal activity, such as the execution of a primitive operation or an intra-component synchronization.

The flow of control of each object is described according to the transitions of the state machine associated to the class of which it is an instance. For each transition

$$l_1 \xrightarrow{[g]t/a} l_2$$

of an abstract state machine we assume a unique intermediate location $l_{1,2}$ to model the interleaving point between the guard and trigger on the one hand, and the action on the other hand. Further, we assume for each boolean *guard* g an evaluation function g such that $g(\sigma, o)$ denotes the boolean result of the evaluation of g by the object o in the object diagram σ; note that guard evaluation is free of side effects, i.e., it does not affect the object diagram itself. Similarly, we assume for each primitive operation a, a state transformer function a such that $a(\sigma, o)$ denotes the object diagram that results from the application of a in the initial diagram σ by the object o. We consider only state transformations that change only instance variables of the object executing it. We do not allow, for example, that an object can assign values to instance variables of other objects within the same component.

The transition relation $\longrightarrow$ associated to a component $\mathcal{C} = \langle B, P \cup R \rangle$ is defined by distinguishing the following cases:

Internal Synchronization: Let o and u be instances of the classes $c, d \in B$, respectively, both inside the component $\mathcal{C}$. Assume the object o is in a location $\sigma(o.loc) = l_1$ while the object u is in the intermediate location $\sigma(u.loc) = l_{3,4}$, where $\sigma(u.x) = o$ and $\sigma(u.y) = v$. If the guard $g(\sigma[o.x/u, o.y/v], o)$ evaluates to true then the synchronization of the objects o and u on the operation op is described by the following rule

$$\frac{l_1 \xrightarrow{[g]op(x,y)/-}_c l_2 \quad l_3 \xrightarrow{-/x.op(self,y)}_d l_4}{\sigma \xrightarrow{\tau} \sigma'},$$

where σ' is the resulting object diagram with $\sigma'(o.x) = u$ and $\sigma(o.y) = v$. The flow of control of the objects o and u is described by their associated state machines and their new locations are $\sigma'(o.loc) = l_{1,2}$, $\sigma'(u.loc) = l_4$, respectively. Note that the evaluation of the guard is in parallel with the execution of the trigger, meaning that the guard g is evaluated in a state that take into account the new values of the actual parameters of the trigger.

Class Instantiation: Let o be an instance of a class $c \in B$. Assume o is in the intermediate location $\sigma(o.loc) = l_{2,3}$ ready to execute a call to the constructor new of the class $d \in B$, with d in the same component of c. If the guard $g(\sigma[u.x/o], u)$ evaluates to true then class instantiation is specified by the following rule

$$\frac{l_0 \xRightarrow{[g]new(x,y)/-}_d l_1 \quad l_2 \xRightarrow{-/d.new(self,x)}_c l_3}{\sigma \xrightarrow{new(o,u)} \sigma'},$$

where l_0 is the initial location of the state machine associated with the class d, and the domain of σ' extends that of σ with the name $u \in Obj(d) \setminus dom(\sigma)$ of the newly created object. The resulting object diagram σ' maps the new name u to the instance variables $o.x$, $u.y$ and $u.self$, while the caller o is assigned to the variable $u.x$. The locations of the two objects o and u are updated to $l_{1,2}$ and l_4, respectively. Finally, all other instance variables of u are set to the undefined value nil.

Primitive Operation: Let o be an object of a class $c \in B$ of the component $\mathcal{C}$ with $\sigma(o.loc) = l_{1,2}$, and let op be a primitive operation. Then

$$\frac{l_1 \xRightarrow{-/op}_c l_2}{\sigma \xrightarrow{\tau} \sigma'},$$

where $\sigma' = op(\sigma, o)[l_2/o.loc]$. The execution of a primitive operation op generates a 'silent' transition transforming the object diagram σ according to the associated function $op(\sigma, o)$ and updating the location loc of the object o to l_2.

Synchronous Operation Call: Let i be a port instance of a role $c \in B$ of the component $\mathcal{C}$, and let $r \in R$ be a required interface of $\mathcal{C}$ declaring the synchronous operation op. Assume that in the object diagram σ the port i is in an intermediate location $\sigma(i.loc) = l_{1,2}$ where it can call a synchronous operation op of the external port $\sigma(i.x) = e$. Then

$$\frac{l_1 \xRightarrow{-/x.op(self,y)}_c l_2}{\sigma \xrightarrow{e.op(i,v)} \sigma'},$$

where $\sigma(i.y) = v$ and σ' is as σ, but for the location loc of i that is assigned to l_2. Note that because x typed by a required interface $r \in R$, there is no class in B with that name. Therefore e is an object not in $dom(\sigma)$.

Constructor Call: A port i instance of a role $c \in B$ of the component $\mathcal{C}$ can create a new port $e \in Obj(r)$ of another component via a call of the constructor new declared in a required interface $r \in R$ of $\mathcal{C}$. This is described by the rule

$$\frac{l_1 \xRightarrow{-/r.new(self,x)}_c l_2}{\sigma \xrightarrow{new(i,e)} \sigma'},$$

where $\sigma(i.loc) = l_{1,2}$, $\sigma'(i.loc) = l_2$ and $\sigma'(i.x) = e$, for some $e \in Obj(r)$. Note that $e \notin dom(\sigma)$, because $r \in R$ is a required interface of $\mathcal{C}$.

Evaluation of a Guard and a Trigger: Let i be a port instance of a role $c \in B$ of the component $\mathcal{C}$, and assume that op is a synchronous operation declared by the provided interface $c \in P$. If in the object diagram σ the port i is in a location $\sigma(i.loc) = l_1$, and the guard $g(\sigma[i.x/e], i)$ evaluates to true, then its trigger op can be executed as

consequence of the reception of the message $op(e, v)$ sent by an external port e. This inter-component synchronization is described by the rule

$$\frac{l_1 \xrightarrow{[g]op(x,y)/-}_c l_2}{\sigma \xrightarrow{i.op(e,v)} \sigma'},$$

where $\sigma'(i.loc) = l_{1,2}$, and $\sigma'(i.x) = e$ and $\sigma'(i.y) = v$ for some value v and object $e \in Obj(d)$ with $d \notin B$.

Port Instantiation: A new instance i of a role $c \in B$ of a component $\mathcal{C}$ can be created by an external port e via a call to the constructor new declared in the provided interface $c \in P$. If the guard $g(\sigma[i.x/e], i)$ evaluates to true, this is described by the rule

$$\frac{l_0 \xrightarrow{[g]new(x,y)/-}_c l_1}{\sigma \xrightarrow{new(e,i)} \sigma'},$$

where $e \in Obj(d)$ with $d \notin B$ and $i \in Obj(c) \setminus dom(\sigma)$ is the identity of the newly created port. Here l_0 is the initial location of the state machine associated to c, and σ' extends σ by assigning $i.loc$ to $l_{0,1}$, $i.self$ and $i.y$ to i, and $i.x$ to e (all other instance variables of i are mapped to the undefined value nil).

Definition 1. *An execution ξ of a component $\mathcal{C}$ is a finite sequence*

$$\sigma_0 \xrightarrow{\ell_1} \sigma_1 \cdots \sigma_{n-1} \xrightarrow{\ell_n} \sigma_n$$

of labelled transitions starting from an initial object diagram σ_0.

From an execution sequence we can extract information about the order of creation among the objects of the component. In fact, given an execution ξ of a component $\mathcal{C} = \langle B, I \rangle$, we define the creation relation $<_\xi$ as the least binary transitive relation on Obj such that

$$o <_\xi u \text{ if } new(o, u) \text{ appears as a label in } \xi,$$

with new the constructor of the class of which u is an instance. Note that in general, the above creation relation will form a forest rather than a tree, because an execution does not record the creation of external ports by other external ports.

3 Testing Semantics

In this section we define a may testing semantics for components. To define the notion of testing semantics, let $ISuccess$ be a distinguished interface consisting of the constructor new and one distinguished operation, $success$, with a parameter of type $ISucess$. We say that a component $\mathcal{C}$ *succeeds*, denoted by $\mathcal{C}\downarrow$, if and only if we may observe only a single call to the $success$ operation by one of its port. More formally, $\mathcal{C}\downarrow$ if and only if there exists an execution ξ of C such that

$$\langle e.success(i, e) \rangle$$

appears as the only communication event in ξ, where e is an external port and i an internal one. This implies that a component may succeed only if $ISuccess$ is one of its required interface.

Definition 2. *Two components $\mathcal{C}_1$ and $\mathcal{C}_2$ with the same provided and required interfaces (not including ISuccess) are may-equivalent, denoted by $\mathcal{C}_1 \simeq \mathcal{C}_2$, if*

$$(\mathcal{C} \oplus \mathcal{C}_1)\downarrow \text{ if and only if } (\mathcal{C} \oplus \mathcal{C}_2)\downarrow$$

for any other component $\mathcal{C}$.

This is a natural adaptation to components of the original definition of may testing semantics for concurrent processes [15]. Note that we allow only the tester component C to require the interface $ISuccess$ and hence to call the $success$ operation by one of its port.

4 Trace Semantics

In the rest of this paper we look for another characterization of the may-equivalence between components that avoids a universal quantification on the tester components. Our starting point are UML message sequence charts. They provide a visual representation of the interactions among of a set of objects in terms of the messages they exchange. Since component interfaces are intended to shield the details of a component implementation from the environment, a sensible semantics for components should abstract from synchronization among objects within the component.

For a given component $\mathcal{C}$, finite sequences of externally observable communication events thus specify the interactions between instances of internal classes realizing the provided interfaces and instances of external classes realizing the required interfaces. Such sequences abstract both from the interactions between instances of classes internal to the components and the interactions between instances of classes external to the component. However, these sequences can be ambiguous or describe information that cannot be implemented by any component. Consider for example the following sequence

$$e.op_1(i, e) \cdot e'.op_2(i, e') \cdot i.op_3(e'', e'')\,,$$

where e, e', and e'' are assumed to be three distinct external ports. The first two events indicate that both e and e' are known to the internal port i, for example because they have been both created by i. In order to justify the last event which involves a call of the operation op_3 of i by e'', there are three possible scenarios:

1. e'' has created i;
2. e'' has received its knowledge of i from e; and
3. e'' has received this knowledge from e'.

These different scenarios are due to three valid assumptions on object creation outside the component, namely e'' can be an ancestor of i,e can be an ancestor of e'', or e' can be an ancestor of e''.

This implicit non-determinism in a sequence of observable events thus allows different incompatible behaviors of the external objects. To resolve this non-determinism we associate to each sequence t of observable events a creation tree.

Definition 3. *A* trace t *is a finite sequence of communication events of the form* $o.op(u,v)$ *together with binary relation* $\prec_t$ *on* Obj *(called the* tree of creation*) such that for each name* u *(but one, the root of the tree) occurring in the sequence there is a unique different name* o *in the same sequence with* $o \prec_t u$.

In the sequel, we denote by $t\restriction_o$ the sub-trace of t with events involving the object o as either the caller or the callee of a synchronous operation. The associated tree of creation is restricted to the names appearing in the restricted sequence (but the root). Moreover, given a component $\mathcal{C} = \langle B, I\rangle$, we denote by $\partial_{\mathcal{C}}(t)$ the result of removing from the trace t all its events that are not externally observable, that is, those communication events involving instances of classes in B as caller or callee of a synchronous operation.

Definition 4. *We define a* trace of a component $\mathcal{C}$ *to be trace* t *consisting of a finite sequence of observable events induced by an execution* ξ *of* $\mathcal{C}$ *together with a creation tree* $\prec_t$ *such that for each ports* o, u *appearing in* t*, if* $o <_\xi u$ *then* $o \prec_t u$.

It should be observed that the creation tree of a trace of a component $\mathcal{C}$ is in fact an abstraction from the actual information on object creation since the latter may involve instances of classes that are strictly internal (or external) to $\mathcal{C}$, i.e., instances of classes that do not realize any provided (or required, respectively) interface. Consequently, the relation $\prec_t$ is more adequately described as the ancestor relation between ports appearing in t that are *indirectly* related because of a creation chain passing through internal objects that do not appear in t.

In general, a trace of a component may still contain impossible events. For example, consider the following execution of a component $\mathcal{C}$

$$\sigma_0 \xrightarrow{new(i,e)} \sigma_1 \xrightarrow{new(i,e')} \sigma_2 \xrightarrow{e.op_1(i,i)} \sigma_3 \xrightarrow{i.op_2(e,e')} \sigma_4$$

inducing the trace t

$$e.op_1(i,i) \cdot i.op_2(e,e')$$

with $i \prec_t e$ and $i \prec_t e'$. The root of the creation tree of t is the internal port i with both the external ports e and e' as children. However, the last communication appearing in t is not possible because the port e cannot possibly know the port e'. To exclude this case, we introduce the following notion of knowledge.

Definition 5. *Given a trace* t*, we define the set* $\kappa(t,o)$ *of objects that an object* o may knows *by induction on* t*:*

$$\kappa(\epsilon, o) = \{o\} \cup \{o' \mid o \prec_t o'\}$$

$$\kappa(t \cdot o'.op(o'',v), o) = \begin{cases} \kappa(t,o) \cup \{o'', v\} & o = o' \text{ and } v \in Obj \\ \kappa(t,o) \cup \{o''\} & o = o' \text{ and } v \notin Obj \\ \kappa(t,o) & \text{otherwise} \end{cases}$$

Intuitively, an object o knows itself, all objects it created, and those objects it received via some triggered operation. The above definition does not depend on a trace

to be generated by an execution of a component. Note however, that given a trace t of a component $\mathcal{C}$ if an external port $e' \in \kappa(t, e)$ then the external port e' *may* also have knowledge of the external port e because an implementation of e and e' may involve the communication of the identity of e to e'. More generally, we can argue in a similar manner that if $e' \in \kappa(t, e)$ then the external objects e and e' may have the same knowledge.

Definition 6. *Given a trace t and a component $\mathcal{C}$, we define a* cluster *of external ports possibly having the same knowledge as an equivalence class of the equivalence relation $\simeq_t$, where $\simeq_t$ is the least equivalence relation such that*

$$e \simeq_t e' \text{ if } e' \in \kappa(t, e)\,.$$

Because objects in a cluster may share their knowledge, we define their shared knowledge $\kappa^*(t, e)$, also called *cluster knowledge*, as

$$\kappa^*(t, e) = \bigcup \{\kappa(t, e') \mid e \simeq_t e'\}\,.$$

We defined clusters only for external ports, because the flow of information of the internal ports is controlled by their respective implementation. For example if i knows e and another external port e' then this in itself does not imply that e may have knowledge of e'. This knowledge can only be obtained by a chain of communications originating from i.

A trace is called executable if external ports communicate only names known by some ports in the same cluster. Formally, we have the following definition.

Definition 7. *Given a component $\mathcal{C}$, a trace t is* executable *if for every prefix $t' \cdot i.op(e, v)$ of t we have that both i and v (if it is an object) are in $\kappa^*(t', e)$. We define $\mathcal{T}(\mathcal{C})$ to be the set of all executable traces of the component $\mathcal{C}$.*

Observe that executable traces are insensitive to the order in which ports are instantiated. Also, because the creation tree of a trace refers only to names that appears in the sequence of observable events (but possibly one, the root), executable traces concerns only with objects that do play a role in an inter-components communication (and not those objects that are created but never used in a communication).

The trace semantics defined above is compositional with respect to component composition.

Theorem 1. *For any two components $\mathcal{C} = \mathcal{C}_1 \oplus \mathcal{C}_2$ we have*

$$\mathcal{T}(\mathcal{C}) = \partial_{\mathcal{C}}(\mathcal{T}(\mathcal{C}_1) \cap \mathcal{T}(\mathcal{C}_2))\,.$$

The proof of this compositionality result involves a fairly straightforward generalization of the compositional trace semantics for CSP (see [9]) to our setting.

The next theorem shows the correctness of the above compositional trace semantics with respect to the above may equivalence.

Theorem 2. *For any components $\mathcal{C}_1$ and $\mathcal{C}_2$ with the same provided and required interfaces (not including ISucess), if $\mathcal{T}(\mathcal{C}_1) = \mathcal{T}(\mathcal{C}_2)$ then $\mathcal{C}_1 \simeq \mathcal{C}_2$.*

The proof of this theorem follows from the compositionality result in Theorem 1 in a fairly standard manner. In the next section we investigate the converse of the above Theorem: are executable traces fully abstract with respect to may equivalence?

4.1 Trace Definability

In order to show that executable traces can be implemented we introduce the notion of extended traces, that is, traces augmented with events for synchronization between external ports, so that they can be justified in terms of what external ports may know.

Definition 8. *An* extended trace t *of an executable trace* t' *of a component* $\mathcal{C}$ *is a trace with the same creation tree of* t' *and that extends the sequence of events of* t' *with additional external communication events of the form* $e.op(e', v)$ *(where* op *may denote a possible operation of an implementation of* e *i.e., an operation that is not specified by the required interface to which* e *belongs).*

In an extended trace the events themselves can be justified directly in terms of the exact knowledge of the ports (i.e. the objects created or received via a triggered operation).

Definition 9. *An* abstract implementation *of an executable trace is an extended trace* t *of an executable trace of a component* $\mathcal{C}$ *such that for every prefix* $t' \cdot o.op(e, v)$ *of* t *both objects* o *and* v *are in* $\kappa(t', e)$.

The following lemma can be proved in a straightforward manner by implementing a protocol for broadcasting new knowledge to all external ports within a cluster.

Lemma 1. *Every executable trace of a component* $\mathcal{C}$ *has an abstract implementation.*

We arrived at the following definability result.

Theorem 3. *For every executable trace* $t \in \mathcal{T}(\mathcal{C})$ *of a component* $\mathcal{C}$ *there exists another component* $\mathcal{C}'$ *with as provided interfaces those required by* $\mathcal{C}$ *and such* t *is also an executable trace of* $\mathcal{C}'$.

The sketch of the proof of the above theorem is as follows. Because t is an executable trace it has an abstract implementation by Theorem 1. Further, we can reduce the latter trace to a sequence s by prefixing it with creation events of the form $new(o, u)$ for each pair of names o and u with $o \prec_t u$, and new the constructor associated to the class of which u is an instance. This way, viewing the creation events above as a binding operator in the second argument, all names occurring in the sequence s are bound but for the root of the tree of creation.

Next, for every external port e in the new sequence s we define an implementation $S(e, s)$ corresponding with the subsequence s of creation and communication events of s involving e. This implementation uses the object names occurring in s as *instance variables* of the object e. Basically, it is constructed by transforming every event $o.op(e, v)$ into a corresponding operation call $o.op(self, v)$, every event $e.op(o, v)$ into a corresponding trigger $op(o, v)$, every creation event $new(e, o)$ into a corresponding constructor call $c.new(self, o)$, with new the constructor of the class c, for $o \in Obj(c)$, and, finally, the every creation $new(o, e)$ into the trigger $new(o, self)$.

As last step, for every required interface r of the component $\mathcal{C}$, we define the UML state-machine specifying the generic behavior of the class realizing the provided interface r of $\mathcal{C}'$ as the *non-deterministic* choice of the implementations $S(e, t)$, where e ranges over all instances of r appearing in t. By construction we have that t is an executable trace of $\mathcal{C}'$.

5 Trace Abstractions

In this section we show that the reverse implication of Theorem 2 does not hold. Therefore executable traces are not fully abstract: there exist may-equivalent components with different sets of executable traces. Moreover, we define trace abstractions for obtaining a fully abstract semantics. We proceed by presenting three typical examples for which full abstraction fails and illustrate the need for respective abstractions on traces.

As a first example, consider a component $\mathcal{C}$ with two required interfaces, r_1 and r_2, both declaring a constructor new. Further, r_1 declares an operation op_1 with a parameter of type r_1, while r_2 declares an operation op_2 with a parameter of type r_1. Let c be a role of the component depending on r_1 and r_2. The transitions of its associated state machine are as follows:

$$l_0 \xrightarrow{/r_1.new(self,x)} l_1 \xrightarrow{/r_2.new(self,y)} l_2 \xrightarrow{/x.op_1(self,x)} l_3 \xrightarrow{/y.op_2(self,y)} l_4$$

Here x is an attribute of type r_1 and y is an attribute of type r_2. Observe that the transition of the above state machine are not guarded and there is no trigger. This state machine generates traces of the form

$$e_1.op_1(i, e_1) \cdot e_2.op_2(i, e_2)$$

with $i \prec e_1$ and $i \prec e_2$, $i \in Obj(c)$, $e_1 \in Obj(r_1)$ and $e_2 \in Obj(r_2)$. Consider now a similar component $\mathcal{C}'$ different from $\mathcal{C}$ in the state machine associated to the class c:

$$l_0 \xrightarrow{/r_1.new(self,x)} l_1 \xrightarrow{/r_2.new(self,y)} l_2 \begin{cases} \xrightarrow{/x.op_1(self,x)} l_{3a} \xrightarrow{/y.op_2(self,y)} l_{4a} \\ \xrightarrow{/y.op_2(self,y)} l_{3b} \xrightarrow{/x.op_1(self,x)} l_{4b} \end{cases}$$

This state machine generates the same traces as the previous one and additionally also traces of the form

$$e_2.op_2(i, e_2) \cdot e_1.op_1(i, e_1)$$

with $i \prec e_1$ and $i \prec e_2$, that differ with the previous ones only with respect to the order of the synchronization on the operations op_1 and op_2. However there is no component that can distinguish these two kinds of traces because the external instances e_1 and e_2 cannot know each other and therefore cannot communicate or synchronize. In other words, the order between these observable events cannot be imposed by the environment because they belong to different clusters.

In general, the order between observable events involving external ports belonging to different clusters cannot be observed in the may-testing semantics. We can abstract from this information by the following closure condition on the traces of a given component.

Definition 10. *Given a component $\mathcal{C}$, a set T of executable traces is closed with respect to the order between events which actively involve external objects belonging to different clusters, if*

$$t \cdot r.op(s, v) \cdot r'.op'(s', v') \cdot t' \in T$$

such that

$$e' \notin \kappa^*(t \cdot r.op(s,v), e),$$

for $e \in \{r, s\}$ *and* $e' \in \{r', s'\}$*, implies*

$$t \cdot r'.op'(s', v') \cdot r.op(s, v) \cdot t' \in T\,.$$

This means that we can only swap events which belong to different clusters of the corresponding prefix of the trace, a phenomena typical of asynchronous processes [6]. In our case, however, this captures the dynamic evolution of clusters, which grow monotonically.

As a second example we consider the following two different state machines associated to a role c (with constructor new_c) of a component depending on a required interface r. This interface declares the constructor new_r and an operation op with a parameter typed by r itself. The first state machine creates an unbounded number of external instances of the required interface r by iteratively calling the constructor method new_r and synchronizes with each of them on the operation op:

$$l_0 \xrightarrow{new_c(x,self)/} l_1 \xrightarrow{/r.new_r(self,y)} l_2 \xrightarrow{/y.op(self,x)} l_1\,.$$

Observe that the iteration is expressed by the fact that, after the call of the operation op, the state machine return in the location l_1. The second state machine implements the above iteration via recursion: it recursively generates an unbounded number of port of c. Each of these ports creates an external instance of the required interface r and synchronize with it via the operation op:

$$l_0 \xrightarrow{new_c(x,self)/r.new_r(self,y)} l_1 \xrightarrow{/y.op(self,y)} l_2 \xrightarrow{/c.new_c(\texttt{self},z)} l_3\,.$$

In term of traces, the component with the first state machine associated to c produces traces of the form

$$e_1.op(i_0, e_1) \cdot e_2.op(i_0, e_2) \cdots e_k.op(i_0, e_k)\,,$$

with $e \prec i_0$ and $i_0 \prec e_n$ for $n = 1, \cdots, k$. On the other hand, the component with the second state machine associated to c produces traces of the form

$$e_1.op(i_0, e_1) \cdot e_2.op(i_1, e_2) \cdots e_k.op(i_{k-1}, e_k)\,,$$

with $e \prec i_0$, $i_{n-1} \prec i_n$ and $i_{n-1} \prec e_n$ for $n = 1, \cdots, k$. Basically the two kinds of traces differ on the identities of the internal ports that create new instances of the required interface r. This difference cannot be observed by another component because each of the external ports e_n's form a different cluster, and objects in different clusters cannot share (and compare) their knowledge.

We can abstract from this difference by, roughly, a cluster-wise renaming of internal instances. Formally, given a component $\mathcal{C}$ we define a relation $t \simeq_\alpha t'$ between the executable traces t and t' if t' results from t by substituting (also in the creation tree) an internal instance i for every occurrence of an other internal instance j, with the same provided interface, in every event which actively involves an external object of

a cluster of t. To preserve the dynamic cluster structure of the internal instances, we additionally require that i does not appear in those events which actively involve an object of the cluster. For example, the first trace above can be obtained by the second one by substituting i_0 for i_{n-1}, with $n = 2, \cdots, k$.

Definition 11. *Given a component $\mathcal{C}$, a set T of executable traces is closed with respect to cluster-wise renaming of internal instances,if*

$$t \in T \text{ and } t \simeq_\alpha t' \text{ implies } t' \in T$$

Finally, we abstract from some information about object creation in a trace t that is too specific, because, after all, the only relevant information concerns the dynamic cluster structure of the trace. Consider the following two traces of a component with a provided interface containing the operation op_p and a required interface containing the operation op_r:

$$e.op_p(i,i) \cdot i.op_r(e',e') \cdot i.op_r(e'',e'')$$

one time with creation tree $i \prec e \prec e' \prec e''$, and another time with creation tree $i \prec e$, $e \prec e'$ and $e \prec e''$. They are two different traces that, however, generate the same cluster structure. In general, the object creator of an instance can be replaced by any other object already existing within the same cluster.

Given a component, we therefore introduce an equivalence relation $t \cong t'$ on executable traces that holds if the traces t and t' specify the same sequence of events with the same dynamic cluster structure, i.e., t and t' have for every prefix the same cluster structure. Formally, a prefix t'' of a trace t consists of a prefix of its sequence of events together with a creation tree obtained by restricting that of t to the objects appearing in t''. So, we define $t_1 \cong t_2$ if for every two prefixes t_1' of t_1 and t_2' of t_2 with the same sequence of observable events σ, we have $o \simeq_{t_1'} u$ if and only if $o \simeq_{t_2'} u$, for every two objects o, u appearing in σ.

Definition 12. *Given a component $\mathcal{C}$, a set T of executable traces is closed with respect to to object creation if*

$$t \in B \text{ and } t \cong t' \text{ implies } t' \in B$$

We have arrived at the following definition of the fully abstract trace semantics $\mathcal{T}_a$ for components.

Definition 13. *Given a component $\mathcal{C}$ we define the set $\mathcal{T}_a(\mathcal{C})$ of its abstract traces as the smallest set of executable traces containing $\mathcal{T}(\mathcal{C})$ and being closed with respect the order between events that actively involve external objects belonging to different clusters, and, the cluster-wise renaming of internal instances.*

Correctness is straightforward because the above closure conditions do not affect may-equivalence.

Theorem 4. *For any components $\mathcal{C}_1$ and $\mathcal{C}_2$, $\mathcal{T}_a(\mathcal{C}_1) = \mathcal{T}_a(\mathcal{C}_2)$ implies $\mathcal{C}_1 \simeq \mathcal{C}_2$.*

6 Full Abstraction

In this section we sketch a proof of full abstraction for the above semantics of components. Full abstraction is expressed by the following theorem.

Theorem 5. *May equivalent components have the same set of abstract traces.*

In the following we give a sketch of the proof that proceeds by contraposition. Suppose $\mathcal{C}_1$ and $\mathcal{C}_2$ are two may-equivalent components with different sets of abstract traces. Without loss of generality, let $t \in \mathcal{T}_a(\mathcal{C}_1) \setminus \mathcal{T}_a(\mathcal{C}_2)$. Since abstract traces are executable, by Theorem 1 there exists an abstract implementation t' of t.

This means that t' contains some protocol for broadcasting new knowledge so that the actual knowledge of external objects coincides with their possible knowledge (details are straightforward and omitted here).

Next we reduce the trace t' to a sequence σ by prefixing it with creation events of the form $new(o, u)$ for each pair of names o and u with $o \prec_{t'} u$, and new the constructor associated to the class of which u is an instance.

We can enrich the sequence σ with additional communication events modelling a protocol for fixing the order of execution among those events of the sequence involving external instances that belong to the same cluster. This protocol can be described using the mechanism of passing a baton between the external instances of the same cluster as in a relay team. Basically we insert between two synchronization events $s_1.op_1(r_1, v_2)$ and $s_2.op_2(r_2, v_2)$ involving two external ports e_1 and e_2 in the same cluster as sender or receiver of the operations, an external event $e_2.baton(e_1, e_1)$. Consequently, the execution of events of instances that belong to the same cluster is sequentialized.

Finally, in order to obtain an observable difference in the may testing semantics, we assume that each cluster of external objects in σ will create an instance o of the provided interface $ICluster$ and call after its last event the operation $cluster$ of o indicating the successful termination of the cluster. As a consequence, there will be as many instances of the class $ICluster$ as actual clusters in the sequence σ. When the last instance is created, an instance of the required interface $ISuccess$ is created and its operation `succ` is called.

On the basis of the above sequence σ, we can construct a distinguishing components $\mathcal{C}$ with as provided interface those required by C_2 plus the interface $ICluster$ and as required interfaces those provided by C_2 plus the interface $ISuccess$. The two interfaces $ISuccess$ and $ICluster$ will be used to indicate the successful termination of all the clusters of external objects of σ. In the state machines associated to the classes realizing the provided interfaces of $\mathcal{C}$ we will use a pseudo-code to describe guards and primitive operations, in particular we will use test for equalities, assignments composed by standard operators like sequential composition ; and if-then-else.

Implementing Abstract Behaviors: First we discuss how to express in pseudo code the abstract behavior of an external instance e in σ. Let $\sigma\restriction_e$ denote the projection of σ onto all the events actively involving the external instance e (as sender, receiver, or creator). Let $\mathcal{R}(\sigma) = \{o_1, \ldots, o_k\}$ be the name space of all the (internal and external) object identities appearing in σ. For notational convenience, we use these object references

also as instance variables in the pseudo code. In order to check for the *local* consistency of the object references stored in the variables of an external instance we introduce for each object reference o a unique fresh variable o' which will be used to store the actual reference received when the object reference o is expected. Let $o \doteq o'$ abbreviate the following pseudo code for for a guard checking the local consistency.

```
if o' = nil
then fail
else  if o ≠ nil
      then  if o ≠ o' then fail fi
      else  for l = 1, ..., k do
                if o' = o_l then fail fi
            od
      fi
fi
```

Here fail is to denote the failure of the evaluation of the guard. This guard first checks whether o' is defined (if o' is undefined the statement aborts because the object reference o is expected). If so, we have two possibilities: either the variable o is already initialized, in which case we simply check whether o equals o', or o is not yet initialized, e.g., not yet received, in which case we check whether o' is different from all the other stored object references.

We can now define a concrete state machine $SM(\sigma\restriction_e)$ describing the abstract behavior of e in σ. For technical convenience we use prefixes of $\sigma\restriction_e$ as locations (with ϵ as initial location and $\sigma\restriction_e$ as final one) and specify the transitions of the state machine by induction on the length of $\sigma\restriction_e$:

$$\sigma'\restriction_e \xrightarrow{/o.op(self,v)} (\sigma' \cdot o.op(e,v))\restriction_e$$

$$\sigma'\restriction_e \xrightarrow{/c.new(self,o)} (\sigma' \cdot new(e,o))\restriction_e$$

$$\sigma'\restriction_e \xrightarrow{[o \doteq o' \text{ and } v \doteq v']op(o,v)/} (\sigma' \cdot e.op(o,v))\restriction_e$$

The state machine $SM(\sigma\restriction_e)$ is thus obtained by a straightforward transformation of the events of $\sigma\restriction_e$ into corresponding actions. The third clause describes the call of a constructor method new which involves the storage of the newly cerated instance in the variable o, with $o \in Obj(c)$. In case of reception of an operation the guards additionally involves a check that the received object references do agree with the corresponding stored ones. Note that thus $SM(\sigma\restriction_e)$ checks only the local consistency of the name space of e. However the encoded protocol for broadcasting new knowledge to all the (external) objects belonging to one cluster will ensure also the global consistency of the name space of the cluster, i.e., any two external objects e and e' belonging to the same cluster assign the same value to any (private) instance variable $o \in \mathcal{R}(\sigma)$. Note however that we cannot ensure that this value is actually the expected object reference o itself!

Implementing the Required Interfaces: For every required interface r of the given component $\mathcal{C}_1$ we can define its implementation as a non-deterministic choice between the state machines $SM(\sigma\restriction_e)$, where e is an instance of r appearing in σ. However, for a full abstraction result, we also need a mechanism which allows such an instance to select its own 'predestined' behavior. The only way we know to implement such a selection is by means of a restricted use of *static class variables*: for each instance e of a required interface r, we introduce a static class variable $r.e$.

Static class variables are variables associated with a class and shared by all its instances only. In languages like Java, static class variables introduce another form of communication besides message passing. Here this means that we associate to each class c a special object with identity c containing the class variables of c. This means that the state transformations associated with primitive operations are not allowed to read and modify the instance variables of the object associated with the class of the instance executing the primitive operation call. In general we want static variables to have no influence on the knowledge of an object (so that two instances of the same class need not necessarily to know each other). This can be enforced by requiring that information stored in static class variables cannot be used in communications between objects, but can only be written and read for private purposes by any instance of a class. More syntactically, we can obtain this by allowing static class variables to appear only in guards (recall that guard evaluation has no side effect) and as parameters of a trigger (so to get assigned to a value). Static variables, however, cannot be communicated, and hence cannot appear neither as parameters of operation calls nor used by a state transformation associated by a primitive operation.

Let $e_1, \ldots e_l$ be the instantiations of r appearing in σ *in that order*. The following state machine with l_0 as initial location allows each instance e_i to select the right location $\sigma\restriction_{e_i}$ where to continue the behavior of the e instance of r by means of a guard preceding the constructor trigger new_r:

$$
l_0 \left\langle \begin{array}{l}
\xrightarrow{[o_1 \doteq o_1' \text{ and } r.e_2 = nil \text{ and } \cdots \text{ and } r.e_l = nil] new_r(o_1, r.e_1)/} \sigma\restriction_{e_1} \\
\xrightarrow{[o_2 \doteq o_2' \text{ and } r.e_1 \neq nil \text{ and } r.e_3 = nil \text{ and } \cdots \text{ and } r.e_l = nil] new_r(o_2, r.e_2)/} \sigma\restriction_{e_1} \\
\vdots \\
\xrightarrow{[o_l \doteq o_l' \text{ and } r.e_1 \neq nil \text{ and } \cdots \text{ and } r.e_{l-1} \neq nil] new_r(o_l, r.e_l)/} \sigma\restriction_{e_l}
\end{array} \right.
$$

Note that static class variables are assigned to the identities of the instances of the class r. Further they do not introduce shared variable concurrency because in the above transitions guard evaluation and trigger (and hence the corresponding test and assignment of static variables) are executed atomically.

May Testing: It still remain to implement the class realizing the provided interfaces $ICluster$ The following state machine is associated to the class realizing the interface $ICluster$ so to ensures that only the last instance of $ICluster$ will create an instance of the interface $ISuccess$, thus indicating that all clusters of external objects have terminated successfully. Again, we use static class variables for the instances of $ICluster$ to 'count' how many instances have already been created (i.e. how many clusters have successfully terminated).

Assuming that the initial trace t contains m clusters of external objects, let $succ_i$, $i = 1, \ldots, m-1$ be $m-1$ static class variables of $ICluster$ (writing for simplicity $succ_i$ instead of $ICluster.succ_i$) in the following state machine associated to it:

$$
l_0 \left\langle \begin{array}{l}
\xrightarrow{[o_1 \doteq o_1' \text{ and } succ_2 = nil \text{ and } \cdots \text{ and } succ_{m-1} = nil] new(o_1, succ_1)/} l_1 \\
\xrightarrow{[o_2 \doteq o_2' \text{ and } succ_1 \neq nil \text{ and } succ_3 = nil \text{ and } \cdots \text{ and } succ_{m-1} = nil] new(o_2, succ_2)/} l_2 \\
\vdots \\
\xrightarrow{[o_m \doteq o_m' \text{ and } succ_1 \neq nil \text{ and } \cdots \text{ and } succ_{m-1} \neq nil] / ISuccess.new(self, x)} l_m
\end{array} \right.
$$

$$
l_m \xrightarrow{/x.success(self,x)} l_{end}
$$

By construction, an instance of $ISuccess$ will be created only after all events of each cluster in the trace t have occurred. Its identity is stored in x and the creator moves in a location from where it calls the operation $success$ of x. Since $ISuccess$ is the only required interface of $C \oplus C_2$, the latter call will generate the only observable event $e.sucess(i, e)$, where $e \in Obj(ISuccess)$ and $i \in Obj(ICluster)$.

Full Abstraction: By construction it follows that $(\mathcal{C} \oplus \mathcal{C}_1)\downarrow$. Furthermore, by construction $(\mathcal{C} \oplus \mathcal{C}_2)\downarrow$ implies $t \in \mathcal{T}_a(\mathcal{C}_2)$. The latter follows basically because the context $\mathcal{C}$ forces $\mathcal{C}_2$ to behave as t up-to the closure conditions.

7 Conclusion and Future Work

We have presented a semantics specification of the behavior of UML-based components that is fully abstract with respect to may equivalence. To focus on the semantic issues involved we have chosen for simplified version of UML class diagrams, object diagrams, state machines and components. However the concepts used are first step towards a semantic approach integrating the several diagrams present in UML. We have applied similar techniques to an extension of the concurrent object calculus with classes [3] and to a sequential object calculus with classes [4]. Both calculi do not consider class inheritance. In fact, and contrary to [16], we do not believe that our result can be applied to an object calculus with inheritance because of the fragile base class problem [21].

Our full abstraction result relies on the static class variables for the construction of the behavior to be associated with a class. The are the key mechanism that allows an object to select its own predestined behavior among those of all instances of a class. Without them we do not know how to construct the behavioral specification of a class from the set of behavior of all its instances. One possibility that we have explored [4] in the context of the object calculus with classes [2], is to restrict it to sequential objects.

The results introduced in this paper are robust enough to support an extension of the state-machine with class name passing, allowing processes to create instances of classes known only at run-time, a form of very late binding typical of component-based systems [22]. Further work is needed for extensions of our result to support more advanced features like inheritance hierarchies, and dynamic class allocation. The first will introduce another way to cross the component borderline, whereas dynamic allocation of behavior to classes (e.g., as studied in [13]) will make this borderline dynamic.

Our fully abstractness result is relevant for and applicable to the generation of test suites for systems of objects. It shows first of all which tests, as sequences of messages, are in fact the same (so it is relevant for defining a effective test suite). Moreover, it shows that to what extent we can abstract from the identities of the test objects. It is future work to apply our result to the theory of testing systems of objects in class based language.

Acknowledgements. Thanks to the anonymous referees and Rocco De Nicola for their comments and suggestions that have improved the paper. This work benefited from discussion with Willem-Paul de Roever and other members of the NWO/DFG bilateral project MobiJ.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. E. Ábrahám, M.M. Bonsangue, F.S. de Boer, and M. Steffen. A Structural Operational Semantics for a Concurrent Class Calculus. Tech. rep. 0307 of the Univ. of Kiel, 2003.
3. M. Steffen, E. Ábrahám, M.M. Bonsangue, F.S. de Boer. Object Connectivity and Full Abstraction for a Concurrent Calculus of Classes In *Proc. ICTAC 2004*, vol 3704 of LNCS, pp. 38-52. Springer, 2005.
4. E. Ábrahám, M.M. Bonsangue, F.S. de Boer, A. Grüner, and M.Steffen. Observability, connectivity, and replay in a sequential calculus of classes. In *Proc. FMCO 2004*, vol. 3657 of LNCS, Springer, 2005.
5. F.S. de Boer, M.M. Bonsangue, and J. Guillen-Scholten. Components: From object to mobile channels. In H. Jifeng and Z. Liu (eds.), *Mathematical Frameworks for Component Software – Models for Analysis and Synthesis*, The World Scientific, 2005.
6. M. Boreale, R. De Nicola, and R. Pugliese. Trace and Testing Equivalence on Asynchronous Processes. *Information and Computation*, 172(2):139-164, 2002.
7. F.S. de Boer and M.M. Bonsangue. A compositional model for confluent dynamic data-flow networks. In *Proc. MFCS*, vol. 1893 of LNCS, Springer 2000.
8. M. Boreale and R. de Nicola. Testing equivalence for mobile processes. *Information and Computation*, 120:279–303, 1995.
9. S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.
10. K. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. The MIT Press, 2002.
11. T. Clark, A. Evans, and E. Kent. The metamodelling language calculus: foundation semantics for UML. In *Proc. FASE 2001*, vol.2029 of LNCS pp. 17–31, Springer 2001.
12. W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A formal semantics of concurrency and communication in Real-Time UML In *Proc. FMCO 2002*, vol. 2582 of LNCS, Springer 2003.
13. S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle II. *ACM ToPLaS* 24(2):153–191, 2002.
14. M. Hennessy. A fully abstract denotational semantics for the π-calculus. *Theoretical Computer Science*, 278(2):53-89, 2002.
15. M. Hennessy and R. de Nicola. Testing equivalence for processes. *Theoretical Computer Science*, 34:83-133, 1984.
16. A. Jeffrey and J. Rathke. A Fully Abstract May Testing Semantics for Concurrent Objects. In *Proc. of the 17th LICS*, pp. 101-112. IEEE Computer Society Press, 2002.

17. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992.
18. Object Management Group, *UML 2.0 Superstructure (Final Adopted specification)*. Document – ptc/03-08-02, August 2004.
19. G. Övergaard Formal Specification of Object-Oriented Meta-Modelling. In *Proc. FASE 2000*, vol. 1783 of LNCS, Springer 2000.
20. B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
21. A. Snyder. Encapsulation and inheritance in object-oriented programming. In *Proc. OOPSLA*, pp. 38–45, SIGPLAN Notices 21:11, 1986.
22. C. Szyperski, D. Gruntz and S. Murer *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, second edition, 2002.

From (Meta) Objects to Aspects: A Java and AspectJ Point of View

Pierre Cointe[1], Hervé Albin-Amiot[1,2], and Simon Denier[1]

[1] OBASCO group, EMN-INRIA, LINA (CNRS FRE 2729),
École des Mines de Nantes, 4 rue Alfred Kastler, La Chantrerie,
44307 Nantes Cedex 3, France
{Pierre.Cointe, Herve.Albin-Amiot, Simon.Denier}@emn.fr
[2] Sodifrance, 4, rue du Château de l'Éraudière, 44324 Nantes, France

Abstract. eWe point some major contributions of the object-oriented approach in the field of separation of concerns and more particularly design-patterns and metaobject protocols. We discuss some limitations of objects focusing on program reusability and scalability. We sketch some intuitions behind the aspect-oriented programming (AOP) approach as a new attempt to deal with separation of concerns by managing scattered and tangled code. In fact AOP provides techniques to represent crosscutting program units such as display, persistency and transport services. Then AOP allows to weave these units with legacy application components to incrementally adapt them. We present a guided tour of `AspectJ` illustrating by examples the new concepts of pointcuts, advices and inter-type declarations. This tour is the opportunity to discuss how the `AspectJ` model answers some of the issues raised by post-object oriented programming but also to enforce the relationship between reflective and aspect-oriented languages.

1 Lessons from Object-Oriented Languages

More than twenty years of industrial practices have clearly enlightened the contributions but also the limitations of object-oriented technologies when dealing with software complexity [15]. Obviously, OO languages have contributed to significant improvements in the field of software engineering and open middleware [30,9]. Nevertheless, programming the network as advocated by `Java` made clear that the object model[33] even extended with the design of reusable micro-architectures such as patterns or frameworks was not enough to deal with critical issues such as scalability, reusability, adaptability and composability of software components [2,19].

In this introduction, we develop some limitations but also two of the main contributions of the OO. approach. These "pro and cons" put together, have challenged new open research ideas in the field of programming languages design and contributed to the emergence of new paradigms such as aspect-oriented programming [19,31].

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 70–94, 2005.
© Springer-Verlag Berlin Heidelberg 2005

1.1 Limitations (CONS)

A first source of problems when programming in the large, is the lack of mechanisms to modularize crosscutting concerns and then to minimize code tangling and code scattering[1]. A second source of problems is the difficulty of representing micro-architectures by using only classes and their methods. A third source of problems is the need of mechanisms to incrementally modify the structure or the behavior of a program. Considering object-oriented programming as THE final technololgy to solve these issues has made clear some well known drawbacks [9,10,15]:

1. Classes schizophrenia: as already quoted by Borning in 1986, classes play too many roles and there is some confusion around the concerns of a class as an object generator, a class as a method dispatcher and an (abstract) class as a part of the inheritance graph.
2. Granularity of behavioral factoring: when expressing behavioral concerns there is no intermediate level of granularity between a method and a class. For instance, there is no way in Java to factorize and then manipulate a set of methods as a whole. Similarly, a Java package - seen as a group of related classes - has not direct manipulable representation at the code level. Then, there is a real need for stateless groups of methods à la trait [28] to implement and compose mixin modules.
3. Class inheritance and transversal concerns: inheritance is not the solution for reusing crosscutting non-functional behaviors such as security or display that are by essence non hierarchical. For instance in `Java`, even very elementary state-less concerns such as being colorable, traceable, memoizable, movable, paintable, clonable, runnable, serializable, ... must be expressed by interfaces to be reused. Unfortunately, these interfaces do not provide any method implementations but only method specifications, limitating reusability.
4. Design patterns traceability: patterns provide reusable micro-architectures based on explicit collaborations between a group of classes [16]. Unfortunately they have no direct representation (reification) at the programming language level raising traceability and understandability issues [18].

1.2 Contributions (PRO)

On the one hand, the *Model View Controller* developed for `Smalltalk` has provided the user with a problem-oriented methodology based on the expression and the combination of (three) separate concerns related to user-interfaces design. The *MVC* was the precursor of *event programming* - in the Java sense - and contributed to make explicit the notion of *join point*, *e.g.,* some well defined points

[1] As stated in [31], *crosscutting* concerns refer to functionalities which do not naturally fit in usual module boundaries, *scattering* can be observed when a functionality must be called from many places and *tangling* when an individual operation may need to refer to many functionalities.

in the execution of a *model* used to dynamically weave the codes associated to the *view* and the *controller*.

On the other hand, object-oriented languages have demonstrated that *reflection* was a general conceptual framework to clearly modularize implementation concerns and to separate them from the functional/business logic. The principle is to introduce a metalevel description and two operations to switch between the base level (user) to this metalevel (implementor). Nevertheless, reflection is solution oriented since it relies on the protocols of the language to build a new solution by opening the system [29].

Our purpose is to develop now those two OO contributions to point later some interesting relationships between objects, design patterns and aspects.

The Model-View-Controller (MVC) was the first attempt to make the notion of concerns explicit when designing the user interface. *MVC* was also the inspirator of the well known `Observer` design pattern (see 3.3).

The main idea was to separate, at the design level, the *model* itself describing the application as a class hierarchy and two separate concerns: the *display*

```
public class Counter extends Object{
  private int value;
  public int getValue(){
    return value;
  }
  public void setValue(int nv){
    value=nv;
    // this.changed();
  }
  public int incr(int delta){
    this.setValue(value+delta);
    return delta;
  }
 public void incr(){
   this.incr(1);
  }
  public void raz(){
    this.setValue(0);
  }
  public String toString(){
    return "@" + value;
  }
  public static void main(String[] args) {
    Counter c1 = new Counter();
    c1.incr(); c1.incr(6); c1.raz();
  }
}
```

Fig. 1. The Counter class

and the *controller*, themselves described as two other separate class hierarchies. At the implementation level, standard object encapsulation and class inheritance were not able to express these crosscutting concerns and not able to provide the coupling between the model, its view, and its controller. This coupling necessitated:

- the introduction (for instance at the root level of the class model) of a *dependence mechanism* in charge of notifying the observers when a source-object state changes. This mechanism is required to automatically update the display when the state of the model changes.
- the instrumentation of (some) methods of the model to raise an event each time the state of the model changed, e.g, each time a given instance variable gets a new value.

To discuss in more details the *MVC* pattern, we transpose for `Java` the well known `Counter` example used by `Smalltalk` teachers. The principle is to develop the model first, as the `Counter` class, and then to introduce its associated `CounterView` and `CounterController` classes.

The Counter Class provides two accessors methods and some basic behaviors such as incrementing, reseting and "stringing" (representing as a text). None of these methods makes any assumption about the views and the controllers used to build the user-interface. Indeed, the associated `CounterView` and `CounterController` classes are defined separately.

Nevertheless, to use this `Counter` class according to the *MVC* paradigm, the developer has to manually manage state changes by inserting a `this.changed()` sentence every time the `value` field receives a new value. If the class is well designed, this insertion can be localized in only one point. In our case, since `incr`, `raz`, ... refer to it, only the setter method `setValue` has to be modified.

The CounterView Class aggregates its `Counter` but also its `CounterController`. The constructor establishes the dependant link between the `Counter` model and its view. In fact, every time the counter executes a `this.changed()`, the view will by notified by receiving an `update()` message. This update will refresh the view by redisplaying the new value of the `Counter` model.

Obviously the first challenge raised by the *MVC* was to automate the transformation of the model and the generation of the associated views and controllers. The second challenge was to proceed this generation in a non invasive way from the model side.

Metalevel Architectures à la Smalltalk and à la CLOS have clearly illustrated the potential of reflection to deal with separation of concerns[30]. The reflective approach makes the assumption that it is possible to separate in a given application, its *why* expressed at the base level, from its *how* expressed at the metalevel.

In the case of a reflective object-oriented language *à la Smalltalk*, the principle is to reify at the metalevel its structural representation, *e.g.,* its classes,

```
public class CounterView extends View {
  private Counter model, CounterController controller;
  private CounterView() {
    model = new Counter();
    model.addDependent(this); // dependency link
  }
  public void update(){
    model.getValue().toString().displayAt(...);
  }
}
public class CounterController extends MouseMenuController {
   private Counter model, CounterView view;
   ...
}
```

Fig. 2. The CounterView and MouseMenuController classes

their methods and the error-messages but also its computational behavior, *e.g.,* the message sending, the object allocation and the class inheritance. Depending on which part of the representation is accessed, reflection is said to be structural or behavioral. Meta-objects protocols (MOPs) are specific protocols describing at the meta-level the behavior of the reified entities. Specializing a given MOP by inheritance, is the standard way [8,17] to extend and to open the base language with new mechanisms such as explict metaclasses [3], multiple-inheritance, concurrency, distribution [24], aspects [4] or reified design patterns.

The design of metaobject protocol such as ObjVlisp, CLOS or ClassTalk contributed to the development of techniques to introspect and to intercess with program structures and behaviors [7,2,26,3]. The minimal *ObjVlisp* model was built upon two classes: `Object` the root of inheritance tree, `Class` the first metaclass and as such the root of the instantiation link, plus `MethodDescription`, the reification of object methods. Then object creation (structural reflection) and message sending (behavioral reflection) can be expressed as two compositions of primitive operations defined in one of these three classes[2]:

- Class.allocate 0 Object.initialize
- Class.lookup 0 MethodDescription.apply

In the case of an open middleware [23], the main usage of behavioral reflection is to control message sending by interposing a metaobject in charge of adding extra behaviors/services (such as transaction, caching, distribution) to its base object. Nevertheless, the introduction of such *interceptors/wrappers* metaobjects requires to instrument the base level with some *hooks* in charge of causally connecting the base object with its metaobject. Those metaobjects prefigured the introduction of AspectJ *crosscuts*, *e.g.,* the specification of execution points where extra actions should be woven in the base program [20,13].

[2] The dot notation `Class.allocate` meaning the `allocate` method defined in the `Class` class.

2 The Java Class Model and Its Associated MOP

The `Java` model is close to the ObjVlisp one, the main difference being that `Class` - the first metaclass - is **final** making `Class` and then the associated class model non extensible [7]. In fact, the Java reflective API[3] provides mainly a self-description of its class architecture and a related MOP mainly dedicated to its introspection.

2.1 Exposing the Java Class Model

On Figure 3, we recognize the `Object` class and the `Class` metaclass. Then the `Constructor`, `Field` and `Method` classes reify the three main kind of Java class members. Each of them implement the `Member` interface and specialize the `AccessibleObject` class introduced to interact with the security manager and get its authorization to intercede with the fields of an object. This simplified figure summarizes also - at the level of every class and using the common *Smalltalk* notation - the names of the metaobject protocols from which we would like to point out:

1. `Class.forName` to reify a class and then to get the reification of its different members,
2. `Class.newInstance` and `Constructor.newInstance` to allocate new objects,
3. `Method.invoke` to call compiled method,
4. `Field.get` and `Field.set` to read/write instance variables.

These methods give access to the description and then the control of some key events associated to OO program execution; respectively object creation, message sending and field references. They can be used as developed below to explicit and then to specialize by inheritance the associated mechanisms.

2.2 Using the Java MOP

ReflectiveObject. One key issue when using reflection is to control the execution flow by monitoring some key events (called hooks [29] or join points [20]) such as field accessing or message sending. The idea is to superimpose additional behaviors before, instead of, or after such events for instance to check pre or post conditions. In that perspective, the Java MOP can be used to explicit and then monitor the accesses to members (field or method) by introducing such hooks to execute extra code. This MOP makes possible the expression of the following rewriting rules expliciting method calls and field references in a *Smalltalk* way:

1. $o.selector(arg_2...arg_n) \hookrightarrow o.receive(\text{“}selector'', arg_2, ...arg_n)$
2. $o.field \hookrightarrow o.instVarAt(\text{“}field'', value)$
3. $o.field = value \hookrightarrow o.instVarAtPut(\text{“}field'', value)$

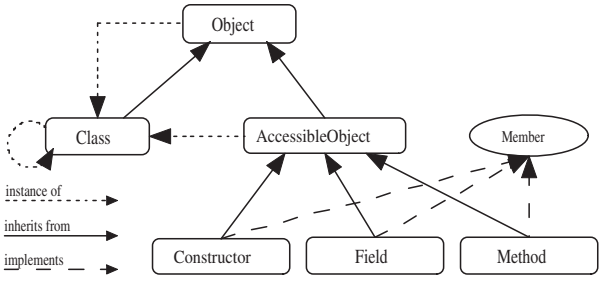
[3] We present here the `java.lang.reflect` package as provided by Java 1.4.

The `ReflectiveObject` class uses the Java MOP to explicit these transformation rules via its `receive` and `instVarAt` methods.

```
public class ReflectiveObject {
  public Object receive(String selector, Object[] args) {
    Method mth = null; Object r = null; Class[] classes = null;
    int lo = 0;
    if (args != null) {
      lo = Array.getLength(args);
      classes = new Class[lo];
     }
    for (int i = 0; i < lo; i++) {classes[i] = args[i].getClass();}
   try {
     // LOOKUP join point
     mth = getClass().getMethod(selector, classes);
    // before method call
     r = mth.invoke(this, args); // APPLY join point
    // after method call
          } catch (Exception e) {System.out.println(e); }
    return r;
  }
  public Object receive(String selector) {
    return receive(selector, null);
  }
  public Object receive(String selector, Object arg1) {
    return receive(selector, new Object[] { arg1 });
  }
  public Object receive(String selector, int arg1) {
    return receive(selector, new Object[] { new Integer(arg1) });
  }
  public void instVarAtPut(String name, Object value) {
    try {
    Field f = this.getClass().getDeclaredField(name);
    if (!Modifier.isStatic(f.getModifiers()))
      f.setAccessible(true);
    // before field reference
      f.set(this, value); // SET reference join point
    // after field reference
    } catch (Exception e) {
       System.out.println("ReflectiveObject.instVarAtPut "+ e);}
  }
}
```

More precisely:

1. `receive` : computes the signature of the associated message by extracting the class of every argument (selector and classes), looking up for an associated method in the class of the receiver (composition of `getClass` and `getMethod`)




```
java.lang.Object(getClass, clone)
    java.lang.Class(getName, newInstance, forName,
                    getDeclaredMethods, getDeclaredFields,
                    getDeclaredConstructors)
    java.lang.reflect.Member(getName, getDeclaringClass, getModifiers)
    java.lang.reflect.AccessibleObject(isAccessible, setAccessible)
       java.lang.Reflect.Field(get, set, setInt)
       java.lang.Reflect.Method(invoke)
       java.lang.Reflect.Constructor(newInstance)
```

Fig. 3. The Java class model and its associated MOP

and then apply this method to its arguments (`invoke`). Obvioulsy, `receive` can be specialized to call extra methods before, around or after the APPLY join point[4].

2. `instVarAtPut`: computes the representation of a field given by its name (composition of `getClass` and `getDeclaredField`) checks if it is an instance or a class variable (`isStatic`), turns the accessible security right to true (`setAccesible`) and change its value (`set`). Consequently, as soon as `Counter` is defined as a subclass of `ReflectiveObject`, its `Counter.setValue` method body can be rewritten as: `this.instVarAtPut (‘‘value’’, new Integer(nv))`. Then, `instVarAtPut` could be overridden to notify the dependents. More generally, this method can be specialized to call extra methods, before, after and around the SET reference join point.

MemoizingObject. The idea is to specialize the previous `receive` method to memoize the already computed results in a cache. For simplification purpose, we made the assumption that `receive` takes only one argument with type `int`. The implementation is as follows; when a `receive` is executed, we extract the first argument `n` and wrap it to a Java `Integer`, we check if the result as already been computed and cached for this argument. If true we directly returns the memoized result. If false, we call the super method in charge of realizing the computation and we record the result:

[4] `receive` allows also the computation of the selector to perform à la Smalltalk the method to call: `new ReflectiveObject().receive(‘‘get’’ + ‘‘class’’)`

```
public class MemoizingObject extends ReflectiveObject {
  public static HashMap cache = new HashMap();
  public Object receive(String selector, Object[] args) {
    int n = ((Integer)args[0]).intValue();
    Integer N = new Integer(n);
    if (cache.containsKey(N)){return (Integer)cache.get(N);}
    else {
      Integer r = (Integer)super.receive(selector, args);
      cache.put(N,r);
      return r;
    }
  }
}
```

The example below develops how to use this memoization concern to compute factorial numbers by the way of the `EFP.fact` method expressed in term of the `MemoizingObject.receive` method:

```
public class EFP extends MemoizingObject {
  static EFP f = new EFP();
  public static int fact(Integer n){
    int pn = n.intValue();
    if (pn == 0)
      return 1;
    else
      return pn * ((Integer)f.receive("fact",
                   (new Integer(pn - 1)))).intValue();
  }
  public static void main(String[] args) {
    try {
      System.out.println("The cache " + memo);
      Integer r= (Integer)f.receive("fact", 4);
      System.out.println("The cache " + memo + r);
      r= (Integer)f.receive("fact", 5);
      System.out.println("The cache " + memo + r);
        } catch (Exception e) {
          System.err.println("MemoizingObject.Main" + e);
        }
  }
}

/* Running EFP.main will produce :
The cache{}
The cache{2=2, 4=24, 1=1, 3=6, 0=1}24
The cache{2=2, 4=24, 1=1, 3=6, 5=120, 0=1}120
*/
```

ClassInspector uses the previous MOP to introspect a given `Java` class and pretty print its "signature". The main idea is to get its direct superclass, its

interfaces and then to enumerate the signature of its different declared members: fields, methods and constructors.

```
public class ClassInspector {
  private java.io.PrintStream out = System.out;

  public void inspect(Class aClass) {
    Field[] fields; Method[] methods; Constructor[] constructors;
    Class[] interfaces = aClass.getInterfaces();
    out.print(aClass.toString() + " extends "
        + aClass.getSuperclass().getName());
    for (int j = 0; j < interfaces.length; j++) {
      out.println("implements" + interfaces[j].getName());
    }
    out.println("");
    out.println("--> declared fields");
    fields = aClass.getDeclaredFields();
    for (int j = 0; j < fields.length; j++)
      out.println("   " + fields[j].toString());
    out.println("--> declared methods");
    methods = aClass.getDeclaredMethods();
    for (int j = 0; j < methods.length; j++)
      out.println("   " + methods[j].toString());
    out.println("--> declared constructors");
    constructors = aClass.getDeclaredConstructors();
    for (int j = 0; j < constructors.length; j++)
      out.println("   " + constructors[j].toString());
  }
  public static void main(String[] args) {
    ClassInspector desc = new ClassInspector();
    try {
      desc.inspect(Class.forName("fmco.Counter"));
    } catch (ClassNotFoundException e) {
      System.err.println(e);
    }
  }
}
```

Annex A.2 gives an example of using `ClassInspector` to "pretty print" our `Counter` class. The purpose is to check the members introduced by the `AspectJ` weaver, as soon as `Counter` is crosscutted by the `CounterObserver` aspect (see 3.3).

2.3 Some Drawbacks of the Java MOP

The previous examples have demonstrated how the `Java` API (via the `Reflective Object` class) can be used to incrementally modify the behavior of a program by controlling message sending or field accessing. Nevertheless the proposed solutions are difficult to generalize since:

- as for the `fact` and `setValue` examples, we need to manually transform regular `Java` codes to explicit the usage of `receive` and `instVarAtPut`,
- contrary to `Smalltalk`, the associated rewriting rules have to deal with `Java` primitive types and necessitates the introduction of casts and wrappers which make the translations more complex,
- `EFP` and `Counter` have to subclass `ReflectiveObject` to get the associated reflective behaviors. Since `Java` only provides single inheritance it is a strong constraint for a class to use this specialization mechanism to get reflective facilities.

At the conceptual level, we can reformulate these drawbacks as the difficulty to systematically reify message sending and field accessing in a non-invasive way and as the problem of modularizing extra code performed "around" such events without using standard class inheritance. At the implementation level, we should discuss the extra execution cost induced by metaprogramming, and the inherent complexity of opening the Java class model [29].

3 A Guided Tour of AspectJ

"A characteristic of aspect-oriented programming, as embodied in AspectJ, is the use of advice to incrementally modify the behavior of a program. An advice declaration specifies an action to be taken whenever some condition arises during the execution of the program. The condition is specified by a formula called a pointcut designator. The events during execution at which advice may be triggered are called joint points. In this model of aspect-oriented programming, join points are dynamic in that they refer to events during the execution of the program [32]."

`AspectJ` is a general purpose language built as a super set of `Java` (see [1] and chaper 6 of [14]). The main idea is to introduce a new unit called an aspect in charge of modularizing crosscutted concerns. This unit looks like a class definition but supports the declaration of pointcuts and advice. These pointcuts are used by a specific compiler to weave the advice with regular `Java` code.

From an industrial perspective, it is the first largely diffused language used to develop or reengineer relevant applications according to aspect-oriented design [14]. From an academic perspective, `AspectJ` is historically the first aspect-oriented language and the natural candidate to expose the relationships between objects, metaobjects and aspects by answering some issues raised by post-object-oriented programming.

3.1 The Join Point and Advice Models

The main intuition behind AOP is to introduce a *join point* model raising events every time an interesting point is reached during the execution of a program. Then the idea is to propose a *pointcut language* to select specific join points and an an *advice language* to express some extra code to be woven at those pointcuts. In the case of `AspectJ` both the pointcut language and the advice language are extensions of `Java`. More precisely and revisiting [19] we propose the following definitions:

- Join point: a well defined point in the execution of a program. As an extension of Java, `AspectJ` proposes about ten different kinds of those points related to object-oriented execution; method call, method execution, field reference (get and set), constructor call, (static) initializer execution, constructor execution, object (pre) initialization, exception handler execution [21].
- Pointcut (*when*): an expression designating a set of join points that optionally exposes some of the values in the associated execution context. These pointcuts can be either user-defined or primitives. These pointcuts can be composed (like predicates) according to three logical operators : logical and (&& operator), logical or (|| operator) and logical negation (! operator).
- Advice (*how/what*): a declaration of what an aspect computes at intercepted join points. In fact a method like mechanism used to declare that certain code should execute when a given pointcut matched. The associated code can be told to run before the actual method starts runing, after the actual method body has run and instead/around the actual method body. Notice that `AspectJ` provides a reification of the current join point by introducting the new `thisJoinPoint` pseudo variable (see the `DummyTrace2` aspect in 3.2).
- Inter-type declaration (introductions): declarations of members that cut across multiple classes or declarations of change in the inheritance relationship between classes. In a reflective way, those declarations are used to open a class by statically introducing new members or by changing its super class or super interfaces.
- Aspect: a modular unit of crosscutting implementation, composed of pointcuts and advice, plus ordinary `Java` member declarations. An `AspectJ` aspect declaration has a form similar to that of a `Java` class declaration.

The rest of this section is a guided tour of `AspectJ` introducing step by step the previous concepts and illustrating them with examples. We will distinguish between behavioral crosscutting affecting the run time behavior and static crosscutting affecting the class and object structures (page 185 of [21]).

3.2 Behavioral Crosscutting

The principle is to modularize simple crosscutting concerns such as monitoring, tracing and memoizing with `AspectJ` aspects and then to adapt existing `Java` classes such as `Counter` and `FP` by weaving their associated advice.

Introducing Pointcuts (Counter and the Daemon aspect). Coming back to the `Counter` class exposed in Figure 1, we still want to notify a dependent (an observer) every time its `value` field is changed. Contrary to the MOP approach, we expect to proceed without editing the definition of the `Counter` class and without evading its `setValue` method. The `AspectJ` solution is to modify automatically and *a posteriori* the code of the `Counter` class by weaving the advice associated to the `Daemon` aspect. This aspect allows to monitor the write access to the `Counter.value` field as follows:

- declaration of a user defined `changed` poincut designator associated to the `Counter.value` set event:
- declaration of an `after` advice which method body will be executed after a set event occur. More precisely after a logical combination of a `changed` **and** an `args` **and** a `target`. The two primitives pointcuts (`args` and `target`) are used to catch - in the dynamic context - the values of the current receiver `r` and the current argument `n`.

```
public aspect Daemon {
  // pointcut designator
  pointcut changed(): set(int Counter.value);
  // after advice
  after(int n, Counter r): changed() && args(n) && target(r){
    r.getDependent().update(n);
  }
}
```

Obviously, poincuts enables to abstract over control flow [25]. In particular, it becomes easy to materialize the execution trace of different method calls. The next three examples develop how to define trace aspects within `AspectJ`.

Introducing Before and After Advice (DummyTrace1). The idea is to define a `DummyTrace` aspect to visualize the computation of the two mutually recursive methods `FP.odd` and `FP.even` are associated to the `FP` (standing for Functional Programming) class. defined in Annex 5.1. The first expected trace looks like the left part of Figure 4. We define the `DummyTrace1` aspect as follows:

- a `traceCounter` field counting the number of the traced method calls. It is used to indent the outputs,
- two `calleven` and `callodd` poincut descriptors monitoring the `FP.even` and `FP.odd` method calls,
- four advice in charge of wrapping these method calls by printing in the standard output the value of the `n` argument **before** the associated pointcut and the value of the same argument plus the computed `b` result `after` the pointcut.

```
-->Before calleven(3)
 -->Before callodd(2)
  -->Before calleven(1)
   -->Before callodd(0)
   false<--After callodd(0)
  false<--After calleven(1)
 false<--After callodd(2)
false<--After calleven(3)
```

```
-->call(FP.even(..)) 3
 -->call(FP.odd(..)) 2
  -->call(FP.even(..)) 1
   -->call(FP.odd(..)) 0
   false<--call(FP.odd(..)) 0
  false<--call(FP.even(..)) 1
 false<--call(FP.odd(..)) 2
false<--call(FP.even(..)) 3
```

Fig. 4. FP class crosscuted by the DummyTrace aspects (DummyTrace1 at left, DummyTrace2 at right)

```
public aspect DummyTrace1 {
  int traceCounter=0;
  void indent(){
    for (int j = 0; j < traceCounter; j++) {System.out.print(" ");}
  }

  pointcut calleven(): call(static boolean FP.even(int)) ;
  pointcut callodd(): call(static boolean FP.odd(int)) ;

  before(int n): calleven() && args(n){
    traceCounter++; indent();
    System.out.println("-->Before calleven(" + n + ")");
  }
  after(int n) returning (boolean b) : calleven()&& args(n){
    indent();
    System.out.println(b + "<--After calleven(" + n + ")");
    traceCounter--;
  }
  after(int n) returning (boolean b) : callodd()&& args(n){
    indent();
    System.out.println(b + "<--After callodd(" + n + ")");
    traceCounter--;
  }
  before(int n): callodd() && args(n){
    traceCounter++; indent();
    System.out.println("-->Before callod(" + n + ")");
  }
}
```

Using Wildcards in Pointcut Signature (DummyTrace2). Our first version of the trace aspect can be improved by i) using only one pointcut designator to declare the methods to be traced and then by ii) factorizing the two before/after advice.

- `AspectJ` supports the use of wild cards in the signatures of pointcuts. The `*` character matches any number of characters and `..` matches zero and more arguments [21]. Consequently the expression `static boolean FP.*(int)` designates all the static methods defined in `FP` taking only one `int` as argument and returning a `boolean`.
- `AspectJ` introduces also the `thisJoinPoint` pseudo-variable to provides reflective dynamic information about the kind of join point, its signature and its context.

By combining wildcard and join point reification, we get a more concise and more generic aspect which corresponding execution trace is given in the right part of Figure 4.

```
public aspect DummyTrace2 {
  int traceCounter=0;
  void indent(){
    for (int j = 0; j < traceCounter; j++) {System.out.print(" ");}
  }
  pointcut boolean_FP_int(): call(static boolean FP.*(int)) ;

  before(int n): boolean_FP_int() && args(n){
    traceCounter++; indent();
    System.out.println("-->" + thisJoinPoint.toShortString() +  " " + n);
  }
  after(int n) returning (boolean b) : boolean_FP_int()&& args(n){
    indent();
    System.out.println(b + "<--" + thisJoinPoint.toShortString() + " " + n);
    traceCounter--;
  }
}
```

Introducing Abstract Aspect (the TraceProtocol Aspect). Since its pointcut descriptor explicitly refers to the `FP` class, the `DummyTrace2` aspect cannot be reused to trace any kind of method in any kind of class. But quoting page 340 of [20]): *"AspectJ provides a simple mechanism of pointcut overriding and advice inheritance. To use this mechanism a programmer defines an abstract aspect, with one or more abstract pointcuts, and with advice on the poincut(s). This, then, is a kind of library that can be parameterized by aspects that extend it".*

To make the trace aspect reusable, we introduce the **abstract** `TraceProtocol` aspect, its associated **abstract** `trace` pointcut descriptor and its two before/after advice:

```
abstract aspect TraceProtocol {
  int traceCounter=0;
  void indent(){
    for (int j = 0; j < traceCounter; j++) {System.out.print(" ");}
  }

  abstract pointcut trace();

  before(int n): trace() && args(n){
    traceCounter++;
    indent();
    System.out.println("-->" + thisJoinPoint.toShortString() +  " " + n);
  }
  after(int n) returning (int r): trace() && args(n){
    indent();
    System.out.println(r + "<--" + thisJoinPoint.toShortString() + " " + n);
    traceCounter--;
  }
}
```

Then we can provide different concrete implementations of this trace protocol. For instance, to trace the `fact` and `fib` static methods also associated to `FP` together with the `incr` method of `Counter` we define the `Tracing` aspect as an extension of `TraceProtocol`:

```
public aspect Tracing extends TraceProtocol {
  pointcut trace() :
    call(int Counter.incr(int))  ||
    call(static int FP.fact(int))  ||
    call(static int FP.fib(int));
}
```

Introducing Around Advice (the Memoizing Aspect). The `around` advice allows to use a join point for adding extra behavior to the proceeding of the standard code execution or by replacing this standard code execution by a totally new code. The `Memoizing` aspect illustrates how to use it in the case of the `FP.fact` static method. As for `MemoizingObject` class in section 2.2, the principle is to memoize all the already computed results in a `cache`. Then when a new call occurs, if the result is already cached then it is directly returned, otherwise the computation is done in a regular way due to the **proceed** construction, the result cached and returned:

```
public aspect Memoizing {
  public static HashMap cache = new HashMap();

  pointcut callfact(): call(static int FP.fact(int)) ;

  int around(int n): callfact() && args(n){
    Integer N = new Integer(n);
    if (cache.containsKey(N)){
      return ((Integer)cache.get(N)).intValue();
    }
    else {
      int r = proceed(n);
      cache.put(N, new Integer(r));
      return r;
    }
  }
}
```

A few enhancements would transform this skeleton aspect into a more generic, multiple methods caching aspect. Compared to the usage of reflection and the definition of the `MemoizingObject` we observe a symetry between the `super` and `proceed` as two constructs allowing to call some overridden behaviors. Obviously the `AspectJ` solution looks better since the definition of the `FP` class is not impacted by the definition of a `Memoizing` concern.

Composing Memoizing and Tracing. `AspectJ` automatically compose different crosscutting aspects. In the case of `FP`, after the definition of the `Memoizing` and `Tracing` aspects and their associated `callfact` and `trace` pointcuts, every call to the `FP.fact` static method will be memoized **and** traced. As shown by Figure 5, after a first call of `FP.fact(3)` (left part) the results of fact(0), fact(1), fact(2) and fact(3) are cached, then when calling `FP.fact(5)`, only the computations of `FP.fact(5)` and `FP.fact(4)` are proceeded (right part):

```
fact(3)
 -->call(FP.fact(..)) 3
  -->call(FP.fact(..)) 2
   -->call(FP.fact(..)) 1
    -->call(FP.fact(..)) 0
    1<--call(FP.fact(..)) 0
   1<--call(FP.fact(..)) 1
  2<--call(FP.fact(..)) 2
 6<--call(FP.fact(..)) 3
fact(3)=6
```

```
fact(5)
 -->call(FP.fact(..)) 5
   -->call(FP.fact(..)) 4
    -->call(FP.fact(..)) 3
    6<--call(FP.fact(..)) 3
   24<--call(FP.fact(..)) 4
 120<--call(FP.fact(..)) 5
fact(5)=120
```

Fig. 5. FP crosscutted by the Memoizing and Tracing aspects

Threading: The Concurrent Aspect. One recurrent question is how to make concurrent the execution of objects. `Java` suggests to use a `Runnable` interface in charge of adapting a class to the `Java` concurrency model. An `AspectJ` alternative programming idiom is to replace the standard execution of a `main` static method by the launching of a new instance of an anonymous `Thread`. Coming back to the clock example discussed in [5] we get:

```
public aspect Concurrent {
  void around() : execution(public static void Clock.main(String[])) {
    new Thread(){
      public void run() {
        System.out.println("Started in another thread");
        proceed();
      }
    }.start();
  }
}
```

3.3 Structural Crosscutting

In the tradition of reflective architecture, `AspectJ` provides a mechanism known as *inter-types declaration* to open `Java` classes (and interfaces) by introducing new members or by changing the inheritance relationship between classes (and interfaces). Quoting the "Introduction to `AspectJ`" available at [1]; *"unlike advice, which operates primarily dynamically, introduction operates statically, at compile-time."*

`AspectJ` supports four kinds of such declarations introductions: field, method and constructor introductions plus class hierarchy alteration. Their syntax is as follows:

1. `modifiers type ClassName.newFieldName [= expression]`
   adds the `newFieldName` to the `ClassName` class with optionally the initial value associated to `expression`,
2. `modifiers type ClassName.newMethodName(parameters) body`
   adds the `newMethodName` to the `ClassName` class with the corresponding `parameters` and method `body`,

3. `modifiers type InterfaceName.addedMethodName(parameters) body` adds the `newMethodName` to **all** the classes implementing the `InterfaceName` class with the corresponding `parameters` and method `body`,
4. `declare parents: ClassName extends SuperClassName` `SuperClassName` becomes the new direct superclass of `ClassName`,
5. `declare parents: ClassName implements ListOfInterfaceNames` `ClassName` implements the new set of `ListOfInterfaceNames`.

In this section, we address the issue of representing the `Observer` design pattern as an aspect. We revisit the presentation of [18] about implementing in `AspectJ` the different design patterns presented in [16]. Obviously, this is an opportunity to reintroduce the `Counter` example and to come back to the `Smalltalk` *MVC*.

The Observer Design Pattern as an Aspect. Quoting [16, page 294], the intent of the `Observer` pattern is to *"define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically"*. The key roles in the `Observer` design patten are `subject` and `observer`. Here we made the assumption of the existence of two interfaces, respectively `Subject` and `Observer`, plus a `Printer` class, all of them defined in Annex A.2. Then the idea is to use a `CounterObserver` aspect to adapt the `Counter` and `Printer` classes to play the roles of `subject` and `observer`.

As shown by Figure 6, the aspect has to adapt the `Counter` class to make it implement the `Subject` interface and in the same time to adapt the `Printer` class to make it implement the `Observer`.

Notice that [12] uses "interface" introductions to address another issue discussed in 1.1 of providing an implementation of traits (set of stateless related methods) in `Java` [28].

The PatternObserverProtocol Aspect. modularizes the update logic and the registration logic for observers. The update logic is handled by the after `stateChanges` advice in charge of updating the list of observers whereas the registration logic is due to a set of introductions. Seven of them are in charge of adding the members `subject`, `getSubject()`, `setSubject()`, `observers`, `addObserver()`, `removeObserver()`, `getObservers()`, to the `Observer` and `Subject` classes. Notice finally how the abstract aspect is parameterized both by the abstract `stateChanges(Subject)` pointcut and the `update` abstract method.

The CounterObserver Aspect. specializes the `PatternObserverProtocol` for a configuration where `Counter` plays the role of a `Subject` and `Printer` the role of an `Observer`. The introduction mechanism is used for this configuration task. First we declare that `Counter` implements `Subject` and `Printer` implements `Observer`. Then we define a concrete `update` method for the `Printer` class. Finally we declare the `stateChanged` pointcut to target and monitor every call of the `Counter.setValue(int)` method.

```
aspect CounterObserver extends PatternOberverProtocol {
  pointcut stateChanges(Subject s):
    target(s) && call(public void Counter.setValue(int));

  declare parents: Counter implements Subject;
  declare parents: Printer implements Observer;

  public void Printer.update() {
    this.print("update occured in Counter" + this.getSubject());
   }
  }
```

Testing the DemoPatternObserver. An important property of this design is the lack of coupling between `Counter`, `Printer` and the `PatternObserverProtocol` abstract aspect. In fact, all coupling are localized in `CounterObserver` and some client such as `Client.main` on Figure 8. Obviously, another aspect can specialize `PatternObserverProtocol` to define its own schema involving other classes. It is one possible reusable implementation of the Observer design pattern: however it is not a universal one. For example, only one instance of the pattern/aspect is allowed per subject, so there is no distinction between non-related observers when updating.

Further Reading About the Join Point Model. This presentation of `AspectJ` is quite short and does not present all the details of its join point model. In particular we do not say anything about so called *scoping join point* matching any join point where the associated code is defined within a given scope or based on the control flow in which they occur. For more details see [1,21].

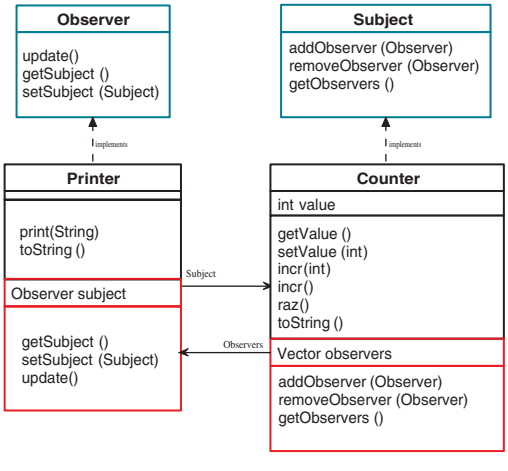


Fig. 6. Counter as a Subject in the Observer design pattern

```
abstract aspect PatternObserverProtocol {
  abstract pointcut stateChanges(Subject s);
  // Update Logic
  after(Subject s, int value): stateChanges(s) && args(value){
    for (int i = 0; i < s.getObservers().size(); i++) {
      ((Observer)s.getObservers().elementAt(i)).update();
    }
  }
  // Registration Logic
  private Vector Subject.observers = new Vector();
  public void Subject.addObserver(Observer obs) {
    observers.addElement(obs);
    obs.setSubject(this);
  }
  public void  Subject.removeObserver(Observer obs) {
    observers.removeElement(obs);
    obs.setSubject(null);
  }
  public Vector Subject.getObservers() { return observers; }
  private Subject Observer.subject = null;
  public void Observer.setSubject(Subject s) { subject = s; }
  public Subject Observer.getSubject() { return subject; }
}
```

Fig. 7. The Observer Design Pattern as an Aspect

```
public static void Client. main(String[] args) {
   Counter c1 = new Counter();
   Printer scribe = new Printer();
   c1.raz();
   scribe.print(c1.toString());
   c1.addObserver(scribe);
   c1.incr(3);
   scribe.print(c1.toString());
   c1.raz();
   scribe.print(c1.toString());
}
/* Will produce
[Printer] @0
 -->call(Counter.incr(..)) 3
[Printer] update occured in Counter@3
 3<--call(Counter.incr(..)) 3
[Printer] @3
[Printer]update occured in Counter@0
[Printer] @0
*/
```

Fig. 8. Counter crosscutted by the CounterObserver and Tracing aspects

4 Conclusion and Open Questions

In this paper we have discussed reflective and aspect-oriented languages two fields of research boosted by the object-oriented community. For pointing out the continuum between objects, metaobjects and aspects we have chosen `Java` and `AspectJ` as the two test beds to express crosscutting concerns in a modular way. Nevertheless, as demonstrated by the proceedings of conferences such as Reflection, ICFP or AOSD (see `http://aosd.net`), meta-entities and aspects are not limited to OO languages but can impact the design of all programming languages including functional, logical and constraint based ones.

In this area, promising open research issues includes providing operational and formal semantics for advice and pointcut models [13,32], using the same general purpose language for defining advices and joint points versus developing multi-paradigm languages, exploring the application of domain specific languages to the definition of aspects, building a reflective kernel as a basis to implement aspect-oriented languages [5,27] and in the opposite direction using AOP as a basis for reflective and metalevel architectures [22], understanding the contribution of aspects to the field of generative programming [6,25], experimenting with the refactoring of legacy codes based on reusable aspects [18].

Acknowledgments

This work is part of the new AOSD network of excellence and its language laboratory (see `http://www.aosd-europe.net`). It benefited from previous discussions presented in [10,11].

References

1. AspectJ site. : See http://eclipse.org/aspectj
2. Aksit, M., Black, A., Cardelli, L., Cointe. P., Guerraoui, R. (editor), and al.: Strategic Research Directions in Object-Oriented Programming, ACM Computing Surveys, volume 8, number 4, page 691-700, (1996).
3. Bouraqadi-Sadani , M.N. , Ledoux, T., Rivard F.: Safe Metaclass Programming. Proceedings of OOPSLA 1998. Editor Craig Chambers,ACM-Sigplan, pages 84-96, volume 33, number 10, Vancouver, British Columbia, USA, October 1998.
4. Bouraqadi-Sadani , M.N. , Ledoux, T.: Supporting AOP Using Reflection. Chapter 12 of [14], pages 261-282, 2005.
5. Chiba, Shigeru.: Generative Programming from a Post Object-Oriented Programming ViewPoint. Proceedings of the Unconventional Programming Paradigms workshop. To appear as LNCS volume. Mont St Michel, France, 2005.
6. Czarnecki, K., Eisenecker, U.W.: Generative Programming. Methods, Tools, and Applications. Addison-Wesley (2000).
7. Cointe, P.: Metaclasses are First Class: The ObjVlisp Model. Proceedings of the second ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 1987). Editor Jerry L. Archibald, ACM SIGPLAN Notices, pages 156-167, volume 22, number 12, Orlando, Florida, USA, October 1987.

8. Cointe, P.: CLOS and Smalltalk : a Comparison. Chapter 9, pages 215-250 of [26]. The MIT Press, 1993.
9. Cointe, P.: Les langages à objets. Technique et Science Informatiques (TSI), volume 19, number 1-2-3, pages 139-146, 2000.
10. Cointe, P., Noyé, J., Douence, R., Ledoux, T., Menaud, J.M., Muller, G., Sudholt, M.: Programmation post-objets. Des langages d'aspect aux langages de composants. RSTI série L'objet. volume 10, number 4, pages 119-143, 2004. See also http://www.lip6.fr/colloque-JFP.
11. Cointe, P.: Towards Generative Programming. Unconventional Programming Paradigms workshop, UPP 04. LNCS 3566, pp 302-312. J.-P. Banâtre et al. Editors. Springer Verlag, 2005.
12. Denier, S.: Traits Programming with AspectJ. RSTI série L'objet. Special issue on Aspect-Oriented Programming (to appear). See also pages 62-78 of the unformal proceeding at `http://www.emn.fr/x-info/obasco/events/jfdlpa04/actes/`, 2005.
13. Douence, R., Motelet, O., Sudholt, M.: A formal definition of crosscuts. Proceedings of the 3rd International Conference on Reflection 2001, LNCS volume 2192, pages 170-186, (2001).
14. Filman, E. R., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley, 2005.
15. Gabriel, R.: Objects Have Failed. See http://www/dreamsongs/com/Essays.html and also http://www.lip6.fr/colloque-JFP/
16. Gamma, E., Helm, R., Johnson. R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. 1995
17. Kiczales, G., Ashley, J., Rodriguez, L., Vahdat, A., Bobrow, D.: Metaobject Protocols Why We Want Them and What Else They Can Do. Chapter 4, pages 101-118 of [26]. The MIT Press, 1993.
18. Hannemann , J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2002. ACM SIGPLAN Notices, volume 37, number 11, pages 161-173.
19. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C, Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. ECOOP 1997 - Object-Oriented Programming - 11th European Conference, volume 1241, pages 220–242. 1997
20. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An Overview of AspectJ ECOOP 2001 - Object-Oriented Programming - 15th European Conference, LNCS volume 2072, pages 327–354. 2001
21. Kiselev, I.: Aspect-Oriented Progamming with AspectJ. Sams Publishing, 2003.
22. Kojarski, S., Lorenz, D., Hirschfeld, R.: Reflective Mechanism in AOP Languages. Draft paper.
23. Ledoux, T.: OpenCorba: A Reflective Open Broker. Proceedings of the second international conference on Meta-Level Architectures and Reflection. Cointe, P.: editor. LNCS 1616, Pages 197-214, Saint-Malo, France, 1999.
24. McAffer, J,: Meta-level Programming with CodA. Proceedings of ECOOP 95. Page 190-214, Springer LNCS, Aarhus, Danemark, 1995
25. Mezini, M., Ostermann, K.: A Comparison of Program Generation with Aspect-Oriented Programming. Proceedings of the Unconventional Programming Paradigms workshop. To appear as LNCS volume. Mont St Michel, France, 2005.
26. Pæpcke, A.: Object-Oriented Programming : The CLOS perspective. The MIT Press, 1993.

27. Rodriguez, L., Tanter, E., Noyé J,: Supporting Dynamic Crosscutting with Partial Behavioral Reflection : a Case Study. RSTI série L'objet. Special issue on Aspect-Oriented Programming (to appear). See also pages 118-137 of the unformal proceeding at `http://www.emn.fr/x-info/obasco/events/jfdlpa04/actes/`, 2005.
28. Scharli, N., Ducasse, S., Nierstrasz, O., Black, P.: Traits: Composable Units of Behaviour. ECOOP 2003 - Object-Oriented Programming - 17th European Conference, Editor L. Cardelli. LNCS volume 2743, pages 248–274. 2003
29. Tanter, E., Noyé, J., Caromel, D., Cointe, P.: Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. Proceedings of the 18th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2003. ACM SIGPLAN Notices, volume 38, number 11, pages 27-46.
30. Thomas, D.: Reflective Software Engineering - From MOPS to AOSD. Journal Of Object Technology, volume 1, number 4, pages 17-26. October 2002.
31. Wand, M.: Understanding Aspects. Invited talk at ICFP 2003. Available at www.ccs.neu.edu/home/wand/ICFP 2003.
32. Wand, M., Kiczales, G., Dutchyn, C.: A semantics for Advice and Dynamic Join Points in AOP. ACM Toplas, volume 26, issue 5, pages 890-910. 2004.
33. Wegner, P.: Dimensions of Object-Based Language Design. Proceedings of the second ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA 1987). Editor Jerry L. Archibald, ACM SIGPLAN Notices, pages 168-182, volume 22, number 12, Orlando, Florida, USA, October 1987.

A Annex

A.1 FP: Our Testbed (Ugly Class)

```
public class FP extends ReflectiveObject{
  public static int fact(int n) {
    if (n == 0)
      return 1;
    else
      return n * fact(n - 1);
  }
  public static int fib(int n) {
    if (n <= 1)
      return 1;
    else
      return fib(n - 1) + fib(n - 2);
  }
   public static boolean even(int n) {
     if (n == 0)
       return true;
     else
       return odd(n - 1);
  }
   public static boolean odd(int n) {
     if (n == 0)
       return false;
```

```
class fmco.Counter extends java.lang.Object implements fmco.Subject
--> declared fields
   private int fmco.Counter.value
   public java.util.Vector fmco.Counter.ajc$...$observers
--> declared methods
   public static void fmco.Counter.main(java.lang.String[])
   public java.lang.String fmco.Counter.toString()
   public int fmco.Counter.getValue()
   public void fmco.Counter.setValue(int)
   public void fmco.Counter.incr()
   public void fmco.Counter.incr(int)
   public void fmco.Counter.raz()
   public void fmco.Counter.addObserver(fmco.Observer)
   public java.util.Vector fmco.Counter.getObservers()
   public void fmco.Counter.removeObserver(fmco.Observer)
--> declared constructors
   public fmco.Counter()
```

Fig. 9. Counter crosscutted by CounterObserver

```
    else
      return even(n - 1);
  }
  public static void main(String[] args) {
    System.out.println("fact(5)=" + fact(5));
    System.out.println("fib(4)=" + fib(4));
  }
}
```

A.2 The Printer Class and the Subject and Observer Interfaces

```
public interface Subject {
  public void addObserver(Observer o );
  public void removeObserver(Observer o);
  public java.util.Vector getObservers();
}
public interface Observer {
  void setSubject(Subject s);
  Subject getSubject();
  void update();
}
public class Printer {
  public void print(String s){
    System.out.println("[Printer] " + s);
  }
  public String toString(){
    return "aPrinter";
```

```
  }
  public static void main(String[] args) {
    new Printer().print("Hello Word");
  }
}
```

A.3 Inspecting Counter Crosscutted by CounterObserver

The reader can check the presence of the `obervers` field as the one of the `addObserver`, `getObservers`, `removeObserver` methods, all of them introduced by the `CounterObserver` aspect.

MoMo: A Modal Logic for Reasoning About Mobility

Rocco De Nicola and Michele Loreti

Dipartimento di Sistemi e Informatica, Università di Firenze,
Viale Morgagni, 65, I-50134 Firenze

Abstract. A temporal logic is proposed as a tool for specifying properties of KLAIM programs. KLAIM is an experimental programming language that supports a programming paradigm where both processes and data can be moved across different computing environments. The language relies on the use of explicit localities. The logic is inspired by Hennessy-Milner Logic (HML) and the $\mu-$calculus, but has novel features that permit dealing with state properties and impact of actions and movements over the different sites. The logic is equipped with a sound and complete tableaux based proof system.

1 Introduction

The increasing use of wide area networks, especially the Internet, has stimulated the introduction of new programming paradigms and languages that model interactions among hosts by means of *mobile agents*; these are programs that are transported and executed on different sites.

In the last decade, several process calculi have been developed to gain a more precise understanding of network awareness and mobility. We mention the Distributed Join-calculus [14], the Distributed π-calculus [20], the Ambient calculus [5], the Seal calculus [9], and Nomadic Pict [25]. The new calculi are equipped with primitives for dealing with the distribution of resources and computational components, for handling code and agent mobility and for coordinating processes. Some of the above mentioned calculi deal also with the key issue of security, namely *privacy* and *integrity* of data, hosts and agents. It is important to prevent malicious agents from accessing private information or modifying private data.

For this class of formalisms, it is thus needed to guarantee sites hosting mobile agents that their privacy is not violated and their data are not modified. Similarly, it is important to guarantee mobile agents that their execution at other sites will not compromise their security. Modal logics have been largely used as formal tools for specifying and verifying properties of concurrent systems. Properties are specified by means of *temporal* and *spatial modalities*. In this paper we advocate the use of modal logics for specifying and verifying dynamic properties of mobile agents running over a wide area network.

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 95–119, 2005.
© Springer-Verlag Berlin Heidelberg 2005

We present a new modal logic that is inspired on one hand by HML (*Hennessy-Milner Logic*) [18] with recursion and on the other hand by the operators of KLAIM (*Kernel Language for Agents Interaction and Mobility*, [11]).

HML is a well-known modal logic used to describe properties of concurrent systems modelled as process algebra terms or as labelled transition systems, its modalities are indexed by the actions that processes can perform.

KLAIM is a language specifically designed to program distributed systems consisting of several mobile components interacting through multiple distributed tuple spaces. Its components, both stationary and mobile, can explicitly refer and control the spatial structures of the network at any point of their evolution. KLAIM primitives allow programmers to distribute and retrieve data and processes to and from the nodes of a net. Moreover, localities are first-class citizens, they can be dynamically created and communicated over the network and are handled via sophisticated scoping rules.

The logics that we describe here can be seen as a simplification of the one presented in [12]. That one is (too) closely related to KLAIM and all its operators are specifically designed for modelling KLAIM systems; its modalities are those dictated by the labels of the KLAIM transition system. The logic that we consider in the current paper aims at being simpler and more general at the same time. It consists of a number of basic operators to be used to describe specific properties/behaviours of mobile and distributed systems. Thus, together with the usual logical connectives and the operators for minimal and maximal fixed points, we have operators for describing dynamic behaviours (*temporal properties*), for modelling resource management (*state properties*), for keeping into account names handling (*nominal properties*), and for controlling mobile processes (*mobility properties*). To have concrete examples of systems properties, we interpret our logical formulae in terms of μKLAIM, a simplified version of KLAIM. However, we do not see much difficulties in using the proposed operators for describing properties of other calculi like π-calculus [23], Dπ [19] or Ambients [5].

Other foundational approaches have adopted logics for the analysis of mobility. In [6] a modal logic for Mobile Ambients has been presented. This very interesting logic is equipped with operators for spatial and temporal properties, for *compositional* specification of systems properties, and specific modal operators are introduced for expressing logical properties of names. However, the main weakness of Ambient logic is that it is not completely decidable; a complete proof system is provided only for a subset of the logic. The main contribution of [6] is the introduction and the analysis of a large set of spatial and temporal modalities for specifying properties of mobile systems. A variant of this logic for asynchronous π-calculus has been presented in [3,4]. This paper introduce a general proof theory for a temporal and spatial logics for mobility.

In [22] and in [10] two variants of Hennessy-Milner Logic for specifying and verifying properties of π-calculus processes have been introduced. The former aims at studying the different equivalences between processes and at understanding the differences between late and early bisimulation. The latter aims at

defining the modal operators that permits describing properties concerning the use, the generation and the transmission of names. Both logics are close related to the π-calculus and cannot be easily generalised to other locality based calculi.

A different approach has been followed in *MobileUnity* [21] and *MobAdtl* [13], two program logics specifically designed to specify and reason about mobile systems by exploiting a Unity-like proof system. These specification languages rely on the use of implementation languages equipped with a formal semantics that permits deriving logical properties of specified systems. The implementation language is not fixed and programmers are left with the problem of guaranteeing the relationship between the implementation and the logical specification. The specification and implementation phase are closely intertwined.

The rest of the paper is organised as follows. In Section 2, we present μKLAIM and its operational semantics. The modal logic is presented in Section 3, while the associated proof system is presented in 4. In Section 5 we exemplify the use of the proposed proof system by considering three simple properties of a simple client-server system that we have previously used as a running example. Section 6 contains some concluding remarks.

2 μKLAIM

KLAIM [11] is a formalism that can be used to model and program mobile systems. It has been designed to provide programmers with primitives for handling physical distribution, scoping and mobility of processes. KLAIM is based on process algebras but makes use of Linda-like asynchronous communication and models distribution via multiple shared tuple spaces [7,16,17]. Tuple spaces and processes are distributed over different localities and the classical Linda operations are indexed with the location of the tuple space they operate on.

For the sake of simplicity, we shall use a simplified version of KLAIM that has been called μKLAIM (see e.g. [1]). The main differences between KLAIM and μKLAIM is that the former allows high-level communication (processes can be used as tuple fields) while the latter only permits evaluating process remotely. Moreover, the simpler language does not make any distinction between physical and logical localities and does not need allocation environments.

2.1 μKLAIM Syntax

A μKLAIM system, called a *net*, is a set of *nodes*, each of which is identified by a *locality*. *Localities* can be seen as the addresses of nodes. We shall use $\mathcal{L}$ to denote the set of localities $l, l_1, \ldots$. Every node has a computational component (a set of processes running in parallel) and a data component (a tuple space). Processes interact with each other either locally or remotely inserting and withdrawing tuples from tuple spaces.

A tuple is a sequence of *actual* fields. Each actual field can be either a locality (l), a value v, from the (finite) set of *basic values Val* (not specified here), or a *variable* x, from the set of *variables Var*. Tuples are retrieved from tuple spaces

Table 1. μKLAIM Syntax

N ::=		NETS	P ::=		PROCESSES
	$\mathbf{0}$	*empty net*		$\mathbf{nil}$	*Empty process*
|	$l :: P$	*located process*	| $act.P$		*Action Prefixing*
|	$l :: \langle t \rangle$	*located tuple*	| $P \mid P$		*Parallel Composition*
|	$N_1 \parallel N_2$	*net composition*	| X		*Recursion Variable*
|	$\nu\, l.N$	*name restriction*	| $\mathbf{rec}X.P$		*Recursion*
			| $\nu\, l.P$		*Name restriction*
act ::=		ACTIONS			
	$\mathbf{out}(t)@l$	*output*	t ::= $f \mid f, t$		TUPLES
|	$\mathbf{in}(T)@l$	*input*	f ::= $l \mid v \mid x$		FIELDS
|	$\mathbf{eval}(P)@l$	*migration*			
			T ::= $F \mid F, T$		TEMPLATES
			F ::= $f \mid !\, l \mid !\, x$		TEMP. FIELDS

via *pattern matching* using *templates* (T). Templates are sequences of *actual* and *formal* fields. The second ones are *variables* that will get a value when a tuple is retrieved. Formal fields are signalled by a '!' before the variable name.

The *pattern-matching* function *match* is defined in Table 2. The meaning of the rules is straightforward: a template matches against a tuple if both have the same number of fields and the corresponding fields do match; two values (localities) match only if they are identical, while formal fields match any value of the same type. A successful matching returns a substitution function (denoted by σ) associating the variables contained in the formal fields of the template with the values contained in the corresponding actual fields of the accessed tuple. Let σ_1 and σ_2 be two substitutions, we use $\sigma_1 \cdot \sigma_2$ to denotes the composition of substitutions:

$$\sigma_1 \cdot \sigma_2(x) = \begin{cases} y & \text{if } \sigma_2(x) = y \\ \sigma_1(x) & \text{otherwise} \end{cases}$$

The syntax of μKLAIM nets is defined in the first part of Table 1. Term $\mathbf{0}$ denotes the *empty net*, i.e. the net that does not contain any node. Terms $l :: P$ (*located process*) and $l :: \langle t \rangle$ (*located tuple*) are used to describe basic μKLAIM nodes: the former states that process P is running at l whilst the latter that the tuple space located at l contains tuple $\langle t \rangle$. μKLAIM nets are obtained by parallel composition ($\parallel$) of located processes and tuples. Finally, $\nu\, l.N$ states that l is a private name within N: We will say that l is private in N.

The following term denotes a net consisting of two nodes, named l_1 and l_2.

$$l_1 :: P_1 \parallel l_1 :: P_2 \parallel l_2 :: (Q_1 | Q_2) \parallel l_2 :: \langle t_1 \rangle \parallel l_2 :: \langle t_2 \rangle$$

Processes P_1 and P_2 are running at l_1 while Q_1 and Q_2 are running at l_2. The tuple space located at l_2 contains tuples $\langle t_1 \rangle$ and $\langle t_2 \rangle$ while that located at l_1 is empty.

The syntax of μKLAIM processes is defined in the second part of Table 1. There $\mathbf{nil}$ stands for the process that cannot perform any actions, $P_1 | P_2$ stands

Table 2. Matching Rules

$$(\mathrm{M}_1)\ \ match(l,l) = [] \qquad (\mathrm{M}_2)\ \ match(!\,l_1, l_2) = \left[l_2/l_1\right]$$

$$(\mathrm{M}_3)\ \ \frac{match(F,l) = \sigma_1 \quad match(T,t) = \sigma_2}{match(\ (F,T)\ ,\ (l,t)\ ) = \sigma_1 \cdot \sigma_2}$$

Table 3. A simple μKLAIM system

$$\begin{array}{l}
Printer :: \mathbf{rec}X.\mathbf{in}(!from)@Printer. \\
\qquad\qquad (X|\mathbf{out}(from)@PrintServer.\mathbf{nil} \\
\\
PrintServer :: \langle PrintSlot\rangle \\
\qquad\qquad | \ \langle PrintSlot\rangle \\
\qquad\qquad | \ \mathbf{rec}X.\mathbf{in}(Print, !from)@PrintServer. \\
\qquad\qquad\ \ (X|\mathbf{out}(from)@Printer. \\
\qquad\qquad\qquad\ \ \mathbf{in}(from)@PrintServer. \\
\qquad\qquad\qquad\ \ \mathbf{out}(PrintOk)@from. \\
\qquad\qquad\qquad\ \ \mathbf{out}(PrintSlot)@PrintServer.\mathbf{nil})
\end{array}$$

for the parallel composition of P_1 and P_2 and $act.P$ stands for the process that executes action act then behaves like P. Possible actions are $\mathbf{out}(t)@l$, $\mathbf{in}(T)@l$ and $\mathbf{eval}(P)@l$.

Action $\mathbf{out}(t)@l$ adds t to the tuple space at locality l. Action, $\mathbf{eval}(P)@l$ spawns a process P at locality l. Action $\mathbf{in}(T)@l$ is used to retrieve tuples from tuple spaces. Differently from **out** this is a blocking operation: The computation is blocked until a tuple matching template T is found in the tuple space located at l. When such a tuple t is found, it is removed from the tuple space and the continuation process is closed with substitution $\sigma = match(T,t)$ that replaces the formals in T with the corresponding values in t. For instance, if $T = (!u, 4)$ and $et = (l, 4)$ then $match(T,t) = [l/u]$. For this reason, $\mathbf{in}(T)@l.P$ acts as a binder for variables in the formal fields of T. Finally, $\nu\, l.P$ declares a new name l that will be used in P: $\nu\, l.N$ and $\nu\, l.P$ act as binders for l in N and P, respectively.

In the rest of the paper, we will use $\tilde{l}$ to denote a finite sequence of names. Moreover, if $\tilde{l} = l_1, \ldots, l_n$, $\nu\, \tilde{l}.N$ will stand for $\nu\, l_1 \ldots \nu\, l_n..N$ while $\{\tilde{l}\}$ will denote the set $\{l_1, \ldots, l_n\}$.

Let γ be a generic syntactic term, we will use $fn(\gamma)$ to denote the set of names free.

Finally, we will write $P_1 =_\alpha P_2$ whenever P_1 is equal to P_2 up to renaming of bound names and variables.

Example 1 (A PrintServer). In Table 3 we show how μKLAIM can be used for modelling a simple print server. We have two μKLAIM nodes: *PrintServer*

and *Printer*. Located at *PrintServer* there is a process that waits for a print request. Each such request contains the locality from which it has been sent. When a request appears in the tuple space at *PrintServer*, the process sends the document to the printer (**out**$(from)@Printer$), waits for the printing signal (**in**$(from)@PrintServer$) and sends an ack (**out**$(PrintOk)@from$) to the client. In order to send a print request a print client has to retrieve a *PrintSlot* from the tuple space located at *PrintServer*. There two *PrintSlot*s are available.

Table 4. μKLAIM operational semantics

$$l_1 :: \mathbf{out}(t)@l_2.P \xrightarrow{l_1 : t \triangleright l_2} l_1 :: P \qquad l_1 :: \mathbf{eval}(Q)@l_2.P \xrightarrow{l_1 : Q \blacktriangleright l_2} l_1 :: P$$

$$\frac{\sigma = match(T, t)}{l_1 :: \mathbf{in}(T)@l_2.P \xrightarrow{l_1 : t \triangleleft l_2} l_1 :: P\{\sigma\}}$$

$$\frac{N_1 \xrightarrow{l_1 : et \triangleright l_2} N_2}{N_1 \parallel l_2 :: P \xrightarrow{\tau} N_2 \parallel l_2 :: P \parallel l_2 :: \langle t \rangle} \qquad \frac{N_1 \xrightarrow{l_1 : Q \blacktriangleright l_2} N_2}{N_1 \parallel l_2 :: P \xrightarrow{\tau} N_2 \parallel l_2 :: P \parallel l_2 :: Q}$$

$$\frac{N_1 \xrightarrow{l_1 : et \triangleleft l_2} N_2}{N_1 \parallel l_2 :: \langle et \rangle \xrightarrow{\tau} N_2 \parallel l_2 :: \mathbf{nil}}$$

$$\frac{N_1 \xrightarrow{\lambda} N_2 \quad l \notin \lambda}{\nu\, l.N_1 \xrightarrow{\lambda} \nu\, l.N_2} \qquad \frac{N_1 \xrightarrow{\lambda} N_2}{N_1 \parallel N \xrightarrow{\lambda} N_2 \parallel N}$$

$$\frac{N_1 \equiv N_1' \quad N_1' \xrightarrow{\lambda} N_2' \quad N_2' \equiv N_2}{N_1 \xrightarrow{\lambda} N_2}$$

Table 5. Structural congruence

$$N_1 \parallel N_2 \equiv N_2 \parallel N_1 \qquad (N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$$

$$l :: P \equiv l :: (P | \mathbf{nil}) \qquad l :: \mathbf{rec}X.P \equiv l :: P[\mathbf{rec}X.P/X]$$

$$l :: (P_1 | P_2) \equiv l :: P_1 \parallel l :: P_2 \qquad \frac{l \notin fn(N_2)}{\nu\, l.(N_1 \parallel N_2) \equiv (\nu\, l.N_1) \parallel N_2}$$

$$N_1 \parallel \mathbf{0} \equiv N_1 \qquad \nu\, l.\mathbf{0} \equiv \mathbf{0}$$

$$\frac{P_1 =_\alpha P_2}{l :: P_1 \equiv l :: P_2} \qquad \frac{l_2 \notin fn(N)}{\nu\, l_1.N \equiv \nu\, l_2.N[l_2/l_1]}$$

2.2 Operational Semantics

In this section we present a labelled operational semantics for μKLAIM where transition labels contain information about the actions performed by located processes.

We let Λ be the set of transition labels λ defined using the following grammar:

$$\lambda ::= \tau \mid l_1 : t \rhd l_2 \mid l_1 : t \lhd l_2 \mid l_1 : P \blacktriangleright l_2$$

Label τ denotes silent transitions while $l_1 : t \rhd l_2$, $l_1 : t \lhd l_2$ and $l_1 : P \blacktriangleright l_2$ denote that a process located at l_1 respectively: inserts tuple t in the tuple space located at l_2; withdraws tuple t from the tuple space located at l_2; or spawns process P to be evaluated at l_2. In the rest of the paper we shall use α to denote transition labels that do not involve process mobility.

The *labelled transition relation*, $\dot{\longrightarrow} \subseteq Net \times \Lambda \times Net$, is the least relation induced by the rules in Table 4 where the structured equivalence of Table 5 is used.

The *structural congruence*, $\equiv$, identifies terms which intuitively represent the same net. It is defined as the smallest congruence relation over nets that satisfies the laws in Table 5. The structural laws express that $\|$ is commutative and associative, α-equivalent processes give rise to equivalent nodes, that the null process and the empty net can always be safely removed/added, that a process identifier can be replaced with the body of its definition, that it is always possible to transform a parallel of co-located processes into a parallel over nodes. Notice that commutativity and associativity of '|' is somehow derived from commutativity and associativity of '$\|$' and from the fact that $l :: (P|Q) \equiv l :: P \parallel l :: Q$.

3 MoMo: A Modal Logic for Mobility

MoMo contains five groups of logical operators that we will describe separately: *kernel fragment*, *state formulae*, *temporal formulae*, *nominal formulae* and *mobility formulae*.

The *kernel fragment* contains standard first-order logic operators and *recursive formulae* that permit describing recursive properties.

State formulae are used to assert properties concerning allocation of resources in the net whilst *temporal formulae* describe how resources are used. Properties of names, like freshness, are specified using *nominal formulae*. Finally, *mobility formulae* describe properties of processes spawned over the net.

In the rest of this section we describe and motivate separately each fragment of the logic and discuss formulae satisfaction. We shall interpret logical formulae over the structure induced by the operational semantics on μKLAIM Nets. The formal syntax and semantics of MoMo will be defined in Section 3.6. In Table 6 we present a set of somehow standard derivable operators that we will use as macro of the logic in the examples in the rest of the paper.

Table 6. Derivable Operators

$\mu\kappa.\phi \stackrel{\text{def}}{=} \neg\nu\kappa.\neg\phi[\neg\kappa/\kappa]$	$\phi_1 \vee \phi_2 \stackrel{\text{def}}{=} \neg(\neg\phi_1 \wedge \neg\phi_2)$
$\rho \Rightarrow \phi \stackrel{\text{def}}{=} \neg\rho \rightarrow \neg\phi$	$[\delta]\,\phi \stackrel{\text{def}}{=} \neg\langle\delta\rangle\neg\phi$
$\forall l.\phi \stackrel{\text{def}}{=} \neg\exists l.\neg\phi$	$\{l_1 \neq l_2\} \stackrel{\text{def}}{=} \neg\{l_1 = l_2\}$
$Always_\delta(\phi) \stackrel{\text{def}}{=} \nu\kappa.\phi \wedge [\delta]\,\kappa\ (\kappa \notin \phi)$	$Never_\delta(\phi) \stackrel{\text{def}}{=} Always_\delta(\neg\phi)$
$Eventually_\delta(\phi) \stackrel{\text{def}}{=} \neg Never_\delta(\phi)$	

3.1 Kernel Fragment

This fragment of the logics is standard. Propositional part of this fragment contains *True* (**T**), *Conjunction* ($\phi_1 \wedge \phi_2$) and *Negation* ($\neg\phi$). The interpretation of this part of the logic is as expected. Every net satisfies *True*, a net satisfies $\phi_1 \wedge \phi_2$ if and only if both ϕ_1 and ϕ_2 are satisfied. Finally, $\neg\phi$ is satisfied by each net that does not satisfy ϕ.

Kernel fragment also contains formulae for specifying recursive properties: *maximal fixed point* and *logical variables*. Intuitively, a net N satisfies $\nu\kappa.\phi$ if N satisfies $\phi[\nu\kappa.\phi/\kappa]$. Formally, the interpretation of $\nu\kappa.\phi$ is defined as the maximal fixed point of the interpretation of ϕ.

If $\mathbb{M}[\![]$ is the interpretation function of the proposed logic (formally defined later), the set of nets satisfying $\nu\kappa.\phi$ is defined as the maximal fixed point of the function $\mathscr{F}_\phi : 2^{Net} \rightarrow 2^{Net}$ defined as follows:

$$\mathscr{F}_\phi(\mathcal{N}) \stackrel{\text{def}}{=} \mathbb{M}[\![\phi]\!][\kappa \mapsto \mathcal{N}]$$

To guarantee well-definedness of the interpretation function it is assumed that in every formula $\nu\kappa.\phi$, logical variable κ is positive in ϕ (i.e. it appears within the scope of an even number of negations). This permits defining $\mathbb{M}[\![\cdot]\!]$ like composition of monotone functions in $2^{Net} \rightarrow 2^{Net}$. Moreover, since 2^{Net} is a complete lattice, we can use Tarski's Fixed Point Theorem that guarantee existence of a unique *maximal* fixed point for $\mathbb{M}[\![\phi]\!]$, when all logical variables in ϕ are positive.

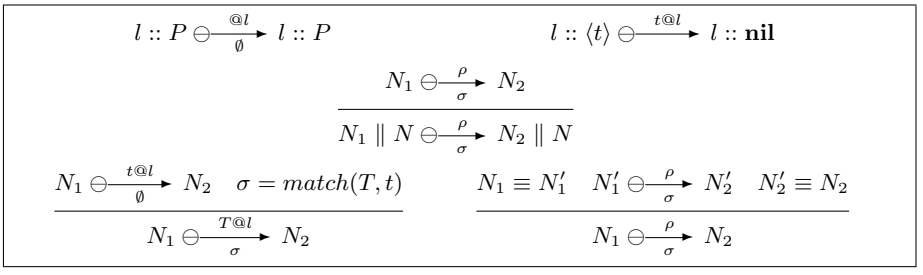
3.2 State Properties

This fragment of the logic aims at describing two categories of properties: *composition properties*, that are used to specify the number of available resources and their distribution over different sites, and *reaction properties*, that describe system reactions to placement of new resources in the net.

To formalise state properties, we have first to specify how to refer *resources*. In μKLAIM one has to consider two kinds of resources: nodes and tuples. To indicate availability of tuples or presence of nodes we let a *resource specification* ρ be one of the following:

- $@l$ indicating existence of a node named l;
- $t@l$ indicating the presence of tuple t at l;
- $T@l$ indicating the presence of a tuple matching template T at l.

Table 7. Consumption Relation



A possible solution to specify distribution of resources could be using just resource specifications as basic formulae. Following this approach a property like *"at l we have either tuple t_1 or tuple t_2"* can be expressed as:

$$t_1@l \vee t_2@l$$

However, this solution would not be completely satisfactory. Because, it would not permit counting the number of available resources. Properties of the form:

$$\textit{"at } l \textit{ we have two tuples matching template } T\textit{"} \qquad \dagger$$

cannot be specified.

For this reason, we introduce the *consumption* operator $\rho \rightarrow \phi$. Intuitively, we have that a net satisfies $\rho \rightarrow \phi$ if after *using* a resource corresponding to ρ the system is in a state where ϕ is satisfied.

A net N satisfies $@l \rightarrow \phi$ if $N \equiv N' \parallel l :: \mathbf{nil}$ and N satisfies ϕ. N satisfies $T@l \rightarrow \phi$ if $N \equiv N' \parallel l :: \langle t \rangle$ and N' satisfies $\phi\sigma$, where $\sigma = match(T, t)$.

To simplify the definition of the semantics of the logic, we introduce relation $\ominus \xrightarrow[\sigma]{\rho}$ (formally defined by the rules of Table 7), where $N_1 \ominus \xrightarrow[\sigma]{\rho} N_2$ if and only if within N_1 a resource described by ρ can be consumed leading to N_2; σ is the substitution induced by the consumption. For instance:

$$l_1 :: \langle l_2 \rangle \ominus \xrightarrow[[l_2/l]]{(!l)@l_1} l_1 :: \mathbf{nil}$$

With this additional relation we have that the satisfaction relation for the consumption operator is:

$$N \models \rho \rightarrow \phi \text{ if and only if } \exists N'.\ N \ominus \xrightarrow[\sigma]{\rho} N' \text{ and } N' \models \phi\sigma$$

With the consumption operator, the property (†) above would be rendered as:

$$T@l \rightarrow T@l \rightarrow \mathbf{T}$$

Production properties are used to state properties depending on the availability of new resources. For instance: *"After inserting tuple t_1 at l_1, eventually l_2 will contain tuple t_2".*

This kind of properties are specified using *production* operator $\rho \leftarrow \phi$, which is satisfied by every net that, after inserting the resource specified by ρ, satisfies ϕ. Hence, a net N satisfies $@l \leftarrow \phi$ if $N \parallel l :: \mathbf{nil}$ satisfies ϕ. N satisfies $t@l \leftarrow \phi$ if $N \parallel l :: \langle t \rangle$ satisfies ϕ.

To model production of a resource described by ρ we let $\overset{\rho}{\oplus\longrightarrow}$ be the relation induced by the rules of Table 8.

Table 8. Production relation

$$N \overset{@l}{\oplus\longrightarrow} N \parallel l :: \mathbf{nil} \qquad N \overset{t@l}{\oplus\longrightarrow} N \parallel l :: \langle t \rangle$$

By using $\overset{\cdot}{\oplus\longrightarrow}$, the satisfaction relation for the production operator is:

$$N \models \rho \leftarrow \phi \text{ if and only if } \exists N'.\ N \overset{\rho}{\oplus\longrightarrow} N' \text{ and } N' \models \phi$$

Please notice that production operator is somehow reminiscent of *separating implication* of *Separation Logic* [24]. This operator permits describing properties of *shared memories* extended with a set of values satisfying a given specification.

Example 2. Let us consider the system presented in Example 1. A possible property to establish is that the print server handles all requests. In other words, one could require that if a client retrieves a print slot and sends a document for printing then sooner or later the document will be printed and the client will be notified. This property can be specified by the following formula:

$$\begin{aligned} \phi \overset{\text{def}}{=} & (PrintSlot)@PrintServer \rightarrow && (1)\\ & @l \leftarrow && (2)\\ & \quad (l, \text{"}test\text{"})@PrintServer \leftarrow && (3)\\ & \quad\quad Eventually_{\tau}(PrintOk@l \rightarrow \mathbf{T}) && (4) \end{aligned}$$

that can be read as:

1. Let $(PrintSlot)$ be a tuple available at $PrintServer$
2. and let l be the locality of a print client,
3. if $(l, \text{"}test\text{"})$ is inserted at $PrintServer$
4. then eventually tuple $(PrintOK)$ will be at l.

3.3 Temporal Properties

The logical operators presented in the previous section permits expressing properties concerning resource availability at specific locations. However, they do not permit describing (constraining) resources usage.

For instance, if we consider the print server of Example 1, the specification of a generic client C located at l would say that:

ClientProp1. C acquires a *print slot* and sends a *print request* at $PrintServer$.
ClientProp2. C never sends a *print request* before acquiring a *print slot*;
ClientProp3. C never reads tuples space located at $PrintServer$

These properties cannot be specified by relying only on state formulae. Indeed, these formulae cannot be used to describe how a given component interacts with the rest of the system.

Temporal properties are specified using operator $\langle\cdot\rangle\phi$. This permits describing properties concerning interaction protocols among components. Modal operator $\langle\rangle$ is indexed either with a transition label α or with $\star$. The former can be τ, $l_1 : t \lhd l_2$ or $l_1 : t \rhd l_2$ (see Section 2.2). The latter, denotes any possible transition label. We shall sometimes use δ to denote either α or $\star$. We will refer to $\langle\alpha\rangle$ as *diamond*, while $\langle\star\rangle$ will be called *next*.

The satisfaction relation for diamond and next operators is:

$$N \models \langle\alpha\rangle\phi \text{ if and only if } \exists N'.\ N \xrightarrow{\alpha} N' \text{ and } N' \models \phi$$
$$N \models \langle\star\rangle\phi \text{ if and only if } \exists \lambda . \exists N'.\ N \xrightarrow{\lambda} N' \text{ and } N' \models \phi$$

Example 3 (PrintClient formalisation). Specifications of desired temporal properties of a print client can be formalised as follows. We assume that the client is located at l.

ClientProp1.

$$\begin{aligned}\phi_1 = [l : \langle PrintSlot\rangle \lhd PrintServer] \\ Eventually_\star(\langle l : Print, l \rhd PrintServer\rangle \mathbf{T})\end{aligned} \tag{1}$$

ClientProp2.

$$\begin{aligned}\phi_2 = \nu\kappa.\ [l : (Print, l) \rhd PrintServer]\,\mathbf{F} \\ \wedge([l : PrintSlot \lhd PrintServer]\,\mathbf{T} \vee [\star]\,\kappa)\end{aligned} \tag{2}$$

ClientProp3.

$$\phi_3 = Never_\star(\langle l : Print, l \rhd PrintServer\rangle \mathbf{T}) \tag{3}$$

3.4 Nominal Properties

In this section we introduce operators for dealing with properties for establishing freshness of names and name quantification. They will enable us to describe properties of the form: *there exists a node in the net that satisfies a given property, two nodes use a restricted (private) location to interact.* Below, we shall consider four different operators: *quantification*, *matching*, *revelation* and *name freshness.*

Name Quantification. $(\exists l.\phi)$ is useful when some names of components in a specified system are not known. For instance, the desired properties of a print client can be rephrased quantifying the locality l. Formally:

$$N \models \exists l.\phi \text{ if and only if } \exists l' \in \mathcal{L}.\ N \models \phi[l'/l]$$

Name Matching. $(\{l_1 = l_2\})$ permits verifying whether two localities correspond to the same node:

$$N \models \{l_1 = l_2\} \text{ if and only if } l_1 = l_2$$

This operator, combined with *name quantification*, permits specifying properties like:

- tuple t can only be located at l_1:

$$Always_\tau(\forall l.\neg(t@l \rightarrow \mathbf{T}) \vee \{l = l_1\})$$

- only processes located at l_1 can read tuple t from the tuple space at l_2

$$Always_\tau(\exists l.\, [l : t \lhd l_2]\, \{l = l_1\})$$

Name Revelation. $(l \circledR \phi)$ means that if l is a restricted name that is uncovered, then formula ϕ holds.

We will say that a net N reveals a locality l $(reveal(N, l))$ if and only if either l is contained in a tuple at a public node of N, or N can perform a transition whose label *reveals* l, i.e. N performs an action at l that operates on a public localities of N, or an action that involves two public localities and *moves* data containing l. Predicate $reveal(N, l)$ is formally defined as follows:

Definition 1. *Let L be a set of localities, N be a net and l be a locality:*

1. *a transition label λ* reveals *l* under *L if and only if $l \notin L$ and:*
   - *either $\lambda = l_1 : t \rhd l_2$, $l = l_1$ or $l \in fn(t)$, and $l_2 \notin L \cup \{l\}$;*
   - *or $\lambda = l_1 : t \lhd l_2$, $l = l_1$, $l \notin fn(t)$ and $l_2 \notin L \cup \{l\}$;*
   - *or $\lambda = l_1 : P \blacktriangleright l_2$ and $l = l_1$ and $l_2 \notin L \cup \{l\}$.*
2. *N* reveals *l $(reveal(N, l))$ if and only if:*
   - *either $N \equiv \nu\, \tilde{l'}.N_1$, $N_1 \xrightarrow{\lambda} N_2$, and λ reveals l under $\{\tilde{l'}\}$,*
   - *or $N \equiv l' :: \langle t \rangle \parallel N'$, $l' \neq l$ and $l \in fn(t)$.*

The satisfaction relation for the name revelation operator is the following:

$$N \models l \circledR \phi \text{ if and only if } \exists N'.N \equiv \nu\, l.N', reveal(N', l') \text{ and } N' \models \phi$$

Name Freshness. $(\diagup\!\!\!\!\!\!\diagup l.\phi)$ acts as a quantifier over all names that do not occur free either in the formula ϕ or in the described system. It is an adaptation of the Gabbay-Pitts quantifiers $\diagup\!\!\!\!\!\!\diagup l.\phi$ [15]. The satisfaction relation for $\diagup\!\!\!\!\!\!\diagup l.\phi$ is:

$$N \models \diagup\!\!\!\!\!\!\diagup l_1.\phi \text{ if and only } \exists l_2.\ l_2 \notin fn(\phi) \cup fn(N) : N \models \phi[l_2/l_1]$$

The fresh name quantifier, when used with state formulae (and in particular with production operator), permits asserting properties related to the creation of new resources.

For instance, we can modify the formula ϕ introduced in Example 2 for specifying that every print request is satisfied in order to guarantee that the location of the print client is *new*: $\diagup\!\!\!\!\!\!\diagup l.\phi$.

3.5 Mobility Properties

The logical operators presented in previous sections do not permit describing properties related to mobility. For instance, properties like "*never a process spawned at l_1 will read tuple t from l_2*" cannot be specified.

To specify this kind of properties we introduce a new modal operator $\langle\rangle.$ that is indexed with a predicate of the form $l_1 : \phi_1 \blacktriangleright l_2$ specifying that: *a process satisfying ϕ_1 is spawned from l_1 at l_2.*

Informally, a net N satisfies $\langle l_1 : \phi_1 \blacktriangleright l_2\rangle.\phi_2$ if and only if formula ϕ_2 is satisfied after a process satisfying ϕ_1 is spawned from l_1 at l_2. In the actual interpretation of ϕ_1 all names shared by the remote evaluated process and the rest of them are considered public. Formally,

$$N \models \langle l_1 : \phi_1 \blacktriangleright l_2\rangle.\phi_2 \text{ if and only if: } N \equiv \nu\,\widetilde{l}.N_1 \parallel l_2 :: \mathbf{nil}\ (l_1, l_2 \notin \widetilde{l}) \text{ and}$$
$$N_1 \parallel l_2 :: \mathbf{nil} \xrightarrow{l_1 : P \blacktriangleright l_2} N_2, l_2 :: P \models \phi_1 \text{ and } \nu\,\widetilde{l}.N_2 \parallel l_2 :: P \models \phi_2$$

We will write $N_1 \xrightarrow{\nu\,\widetilde{l}.l_1 : P \blacktriangleright l_2} N_2$ if and only if $N_1 \equiv \nu\,\widetilde{l}.N' \parallel l_2 :: \mathbf{nil}$ $(l_1, l_2 \notin \{\widetilde{l}\})$ and $N' \xrightarrow{l_1 : P \blacktriangleright l_2} N_2$.

The previously considered property "*never a process evaluated at l_1 will read tuple t from l_2*", can be specified as follows:

$$Never_\tau(\langle l_1 : Eventually_\star(\langle l_1 : t \lhd l_2\rangle \mathbf{T}) \blacktriangleright l_2\rangle.\mathbf{T})$$

Table 9. Syntax of Formulae

$\phi ::=$	Formulae
$\mathbf{T}$	*true*
$\phi \wedge \phi$	*conjunction*
$\neg\phi$	*negation*
$\nu\kappa.\phi$	*maximal fixed point*
κ	*logical variable*
$\rho \rightarrow \phi$	*consumption*
$\rho \leftarrow \phi$	*production*
$\langle\star\rangle\phi_2$	*next*
$\langle\alpha\rangle\phi_2$	*diamond*
$\exists l.\phi$	*name quantification*
$\{l_1 = l_2\}$	*matching*
$l \circledR \phi$	*revelation*
$\not\exists l.\phi$	*fresh name quantification*
$\langle l_1 : \phi_1 \blacktriangleright l_2\rangle.\phi_2$	*Mobility*

Table 10. Interpretation of Formulae

$$
\begin{array}{l}
\mathbb{M}[\![\mathbf{T}]\!]\varepsilon = Net \\
\mathbb{M}[\![\neg\phi]\!]\varepsilon = Net - \mathbb{M}[\![\phi]\!]\varepsilon \\
\mathbb{M}[\![\phi_1 \vee \phi_2]\!]\varepsilon = \mathbb{M}[\![\phi_1]\!]\varepsilon \cup \mathbb{M}[\![\phi_2]\!]\varepsilon \\
\mathbb{M}[\![\rho \rightarrow \phi]\!]\varepsilon = \{N | N \ominus^{\rho}_{\sigma} N' \text{ and } N' \in \mathbb{M}[\![\phi\sigma]\!]\varepsilon\} \\
\mathbb{M}[\![\rho \leftarrow \phi]\!]\varepsilon = \{N | N \oplus^{\rho} N' \text{ and } N' \in \mathbb{M}[\![\phi]\!]\varepsilon\} \\
\mathbb{M}[\![\langle\alpha\rangle\phi]\!]\varepsilon = \{N | N \xrightarrow{\alpha} N' \text{ and } N' \in \mathbb{M}[\![\phi]\!]\varepsilon\} \\
\mathbb{M}[\![\langle\star\rangle\phi]\!]\varepsilon = \{N | N \xrightarrow{\lambda} N' \text{ and } N' \in \mathbb{M}[\![\phi]\!]\varepsilon\} \\
\mathbb{M}[\![\nu\kappa.\phi]\!]\varepsilon = \bigcup\{\mathcal{N} | \mathcal{N} \subseteq \mathbb{M}[\![\phi]\!]\varepsilon \cdot [\kappa \mapsto \mathcal{N}]\} \\
\mathbb{M}[\![\kappa]\!]\varepsilon = \varepsilon(\kappa) \\
\mathbb{M}[\![\exists l.\phi]\!]\varepsilon = \bigcup_{l' \in \mathcal{L}} \mathbb{M}[\![\phi[l'/l]]\!]\varepsilon \\
\mathbb{M}[\![\{l_1 = l_2\}]\!]\varepsilon = \begin{cases} Net & \text{if } l_1 = l_2 \\ \emptyset & \text{otherwise} \end{cases} \\
\mathbb{M}[\![l \circledR \phi]\!]\varepsilon = \{N | N \equiv \nu l.N', reveal(N', l) \text{ and } N' \in \mathbb{M}[\![\phi]\!]\varepsilon\} \\
\mathbb{M}[\![/l_1.\phi]\!]\varepsilon = \{N | \exists l_2.l_2 \notin fn(\phi) \cup fn(N) \text{ and } N \in \mathbb{M}[\![\phi[l_2/l_1]]\!]\varepsilon\} \\
\mathbb{M}[\![\langle l_1 : \phi_1 \blacktriangleright l_2\rangle.\phi_2]\!]\varepsilon = \{N | N \xrightarrow{\nu\,\tilde{l}.l_1:P \blacktriangleright l_2} N_2 : \\
\qquad\qquad l_2 :: P \in \mathbb{M}[\![\phi_1]\!]\varepsilon \text{ and } \nu\,\tilde{l}.N_2 \parallel l_2 :: P \in \mathbb{M}[\![\phi_2]\!]\varepsilon\}
\end{array}
$$

3.6 Syntax and Semantics of MoMo

In this section we formally define syntax and semantics of MoMo logic. The actual syntax is summarised in Table 9 while the interpretation function is formally defined in Table 10.

Interpretation function $\mathbb{M}[\![]\!]$ takes a formula ϕ and a *logical environment* and yields the set of nets satisfying ϕ or, equivalently, the set of nets that are *models* of ϕ. A logical environment, denoted by ε, is a function that for each logical variable yields a set of μKLAIM nets. We use ε_0 to denote the logical environment such that, for each logical variable κ, $\varepsilon_0(\kappa) = \emptyset$.

We shall use $\mathbb{L}$ to denote the set of MoMo formulae. Moreover, in the rest of the paper, we will write $\phi_1 \prec \phi_2$ to denote that ϕ_1 is a *proper subformula* of ϕ_2.

The following lemma guarantees that names can be consistently replaced in nets and formulas preserving the satisfaction relation.

Lemma 1. *For every net N, formula ϕ and localities l_1 and l_2: $N \in \mathbb{M}[\![\phi]\!]\varepsilon$ if and only if $N[l_2/l_1] \in \mathbb{M}[\![\phi[l_2/l_1]]\!]$.*

4 Proof System

To verify whether a concurrent system satisfies a formula two main approaches are classically available. These are respectively based on *proof systems* and *model checking*. In the proof system approach, inference rules are provided in order to build proofs that establish formulae satisfaction. In the model checking approach, instead, a methodology is introduced to verify *automatically* whether a system is a model of a formula.

In this section, we introduce a *so called* tableau based proof system for our logic. Tableau systems are finite families of deduction rules which sanction reduction of goals to subgoals. They are similar in style to structured operational semantics. With tableau one can systematically generate subcases until elementary contradictions/assertions are reached.

Now, we briefly introduce some basic notions about derivation trees and their component, then we describe the proof system for our logic by introducing specific set of rules for each of its.

4.1 Sequents and Proofs

The proof system operates on *sequents* (denoted by π, π_1, π_2, ...) of the form $H \vdash N : \phi$ where N is a net, ϕ is a formula and H is a set of *hypothesis*, i.e. a set of pairs of the form $N_i : \phi_i$.

Proof system derivation rules have the following form:

$$\frac{\{\pi_1, \ldots, \pi_n\}}{\pi}\ cond$$

where $\{\pi_1, \ldots, \pi_n\}$ is the (finite) set sequents to prove in order to assess *validity* of π and *cond* is a side condition. A rule like the above can be *applied* only when the side condition is satisfied. Δ will be used to denote a derivation (a proof) within the proof system.

A derivation Δ is a *derivation from* π if and only if it has π as root and it is maximal, in the sense that nothing can be derived from its *leaves*.

A sequent $\pi = H \vdash N : \phi$ is *successful* when one of the following conditions holds:

- $\phi = \mathbf{T}$;
- $\phi = \{l = l\}$;
- there exists N' such that $N' \equiv N$ and $(N' : \phi) \in H$;
- sequent π is derivable from $\emptyset$.

A derivation Δ is *successful* if all its leaves are successful. If Δ is a successful derivation from π, then Δ is a *proof* for π. A sequent π is provable if there exists a derivation Δ that is a proof for π. We will say that N *satisfies* ϕ *under the assumptions* H if $H \vdash N : \phi$ is provable.

4.2 Names Handling

To manage name quantification while guaranteeing finiteness of derivations, we need to introduce a mechanism for selecting new localities. For this reason, we assume that the set of localities $\mathcal{L}$ is totally ordered by the ordering relation $\leq_{\mathcal{L}}$ with l_0 as minimal element. Moreover, if L is a subset of $\mathcal{L}$, we let $sup(L)$ be the locality not in L that is a least upper bound for the elements in L.

Similarly, we need to set up a machinery for handling all the transitions a net can perform. In principle, these could be infinite. Indeed, we have that if

$Next(N)$ is the set of pairs (λ, N') such that $N \xrightarrow{\lambda} N'$ (see Table 4), we have no guarantee that $Next(N)$ is finite. For instance, if we consider the net:

$$N_1 = l_1 :: \mathbf{in}(!l)@l_1.\mathbf{out}(l_1)@l.\mathbf{nil}$$

we have $Next(N_1) = \left\{(l_1 : l' \lhd l_1, l_1 :: \mathbf{out}(l_1)@l'.\mathbf{nil}) \;\middle|\; l' \in \mathcal{L}\right\}$ that is infinite because $\mathcal{L}$ is such.

A solution to this problem would be considering a finite representation of all the possible transitions up to renaming. This would enable us to consider, for any net N and any formula ϕ, only transitions whose label λ contains *interesting* localities that are either free names of N and ϕ or names taken from a finite set of localities that come immediately after them according to $\leq_{\mathcal{L}}$. Now we have that the cardinality of the set from which localities are taken is equal to the (finite) number of localities that are free in the net (and in the analysed formula) plus the cardinality of the (finite) set of free localities in λ. This intuition can be formalised as follows:

Definition 2. *Let L be a set of localities and λ be a transition label, we say that λ is interesting for L, notationally $\lambda \Uparrow L$, if $fn(\lambda) \subseteq \mathcal{E}_i(L)$. Where $i = |fn(\lambda)|$ and $\mathcal{E}_i(L)$ is inductively defined as follows:*

- $\mathcal{E}_0(L) = L$;
- $\mathcal{E}_{i+1}(L) = \mathcal{E}_i(L \cup \{sup(L)\})$.

For net N_1, described above, when considering the satisfaction of formula $[\star]\,\mathbf{T}$, we would have that:

$$Next(N_1) = \{(l_1 : l_1 \lhd l_1, l_1 :: \mathbf{out}(l_1)@l_1.\mathbf{nil}), (l_1 : l_2 \lhd l_1, l_1 :: \mathbf{out}(l_2)@l_1.\mathbf{nil})\}$$

where l_2 is equal to $sup(fn(N_1) \cup fn([\star]\,\mathbf{T})) = sup(\{l_1\})$.

The following lemma ensures that, at each step of a proof, only a finite set of possible *next* configurations can be take into account.

Lemma 2. *Let N be a net and L a finite set of localities such that $fn(N) \subseteq L$ the set $\left\{(\lambda, N') \;\middle|\; N \xrightarrow{\lambda} N' \text{ and } \alpha \Uparrow L\right\}$ is finite.*

4.3 Proof Rules

The rules of the proof system for our logic are presented in Tables 11–17. For each operator of the logic the proposed proof system provides two rules; one for the operator and one for its negation.

Propositional Fragment. The proof rules for the propositional fragment of the logic are presented in Table 11, and are somehow exptected. To prove $H \vdash N : \phi_1 \wedge \phi_2$ both $H \vdash N : \phi_1$ and $H \vdash N : \phi_2$ have to be proved. A proof for $H \vdash N : \neg(\phi_1 \wedge \phi_2)$ can derived either from $H \vdash N : \neg\phi_1$ or from $H \vdash N : \neg\phi_2$. Double negation can be removed in order to prove $H \vdash N : \neg\neg\phi$.

Table 11. Proof System: Propositional Fragment

$$\frac{H \vdash N : \phi_1 \quad H \vdash N : \phi_2}{H \vdash N : \phi_1 \wedge \phi_2} \qquad \frac{H \vdash N : \neg\phi_i}{H \vdash N : \neg(\phi_1 \wedge \phi_2)}$$

$$\frac{H \vdash N : \phi}{H \vdash N : \neg\neg\phi}$$

Recursive Formulae. The rules for recursive formulae are formally defined in Table 12. They rely on the *proper subformula* relation ($\phi_1 \prec \phi_2$), introduced in Section 3.6, and on the *hypothesis discharging* technique at the bottom of Table 12. We let $H \downarrow \phi$ be the set of hypotheses obtained from H by removing all the formulae containing ϕ as proper subformula.

Table 12. Proof System: Recursive Formulae

$$\frac{H \downarrow \nu\kappa.\phi \cup N : \nu\kappa.\phi \vdash N : \phi[\nu\kappa.\phi/\kappa]}{H \vdash N : \nu\kappa.\phi} \nexists N' : N \equiv N', (N' : \phi) \in H$$

$$\frac{H \downarrow \nu\kappa.\phi \cup N : \nu\kappa.\phi \vdash N : \neg\phi[\nu\kappa.\phi/\kappa]}{H \vdash N : \neg\nu\kappa.\phi} \nexists N' : N \equiv N', (N' : \phi) \in H$$

where $$H \downarrow \phi = H - \{N' : \phi' | \phi \prec \phi'\}$$

In order to prove that N satisfies $\nu\kappa.\phi$ under H, two cases have to be considered. If there exists N' such that $N \equiv N'$ and $N' : \nu\kappa\phi \in H$, then $H \vdash N : \nu\kappa.\phi$ is a successful sequent and the wanted result is established. Otherwise, a proof for $H \vdash N : \nu\kappa.\phi$ can be obtained by establishing satisfaction of the formula obtained by unfolding $\nu\kappa.\phi$, i.e. $\phi[\nu\kappa.\phi/\kappa]$, under the assumption that N satisfies $\nu\kappa.\phi$. To avoid interferences between different hypotheses, all formulae that contains $\nu\kappa.\phi$ as a proper subformula are removed (*discharged*) from H. These formulae do not play any role in determining whether N satisfies $\nu\kappa.\phi$. However, when $\nu\kappa.\phi$ is unfolded to $\phi[\nu\kappa.\phi/\kappa]$, some of the assumptions may play an improper role in the proof for $H \vdash N : \phi[\nu\kappa.\phi/\kappa]$.

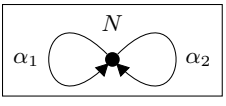
To see how missing the discharging of a formula can lead to problems, let us consider the derivation of Table 13. There it is proved that net N, whose transitions are described in the accompanying figure, satisfies $\neg\phi_A$, where:

$$\phi_A = \nu\kappa_1.\neg\langle\alpha_1\rangle(\neg\kappa_1 \vee \phi_B) \tag{4}$$

$$\phi_B = \nu\kappa_2.\langle\alpha_2\rangle\neg\nu\kappa_1.\neg\langle\alpha_1\rangle(\neg\kappa_1 \vee \phi_B) \tag{5}$$

The formula asserts that it is possible to perform infinitively often a any number of transitions labelled α_1 and then a transition labelled α_2. The deriva-

Table 13. A derivation



$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{N:\phi_B, N:\phi_A \vdash N:\phi_B}{N:\phi_B, N:\phi_A \vdash N:\neg\phi_A \vee \phi_B}\,(10)}{N:\phi_B, N:\phi_A \vdash N:\langle\alpha_1\rangle(\neg\phi_A \vee \phi_B)}\,(9)}{N:\phi_B, N:\phi_A \vdash N:\neg\neg\langle\alpha_1\rangle(\neg\phi_A \vee \phi_B)}\,(8)}{N:\phi_B \vdash N:\neg\phi_A}\,(7)}{N:\phi_B \vdash N:\langle\alpha_2\rangle\neg\phi_A}\,(6)}{N:\phi_A \vdash N:\phi_B}\,(5)}{N:\phi_A \vdash N:\neg\phi_A \vee \phi_B}\,(4)}{N:\phi_A \vdash N:\langle\alpha_1\rangle(\neg\phi_A \vee \phi_B)}\,(3)}{N:\phi_A \vdash N:\neg\neg\langle\alpha_1\rangle(\neg\phi_A \vee \phi_B)}\,(2)}{\vdash N:\neg\phi_A}\,(1)$$

tion is successful since the last sequent amounts to requiring to prove that N satisfies ϕ_B when is assumed that $N : \phi_B$.

If $N : \phi_A$ was not discharged at step (5), no successful derivation for $\vdash N : \neg\phi_A$ would exist, and the proof system would be unsound. Indeed, rule of step (7) could not be applied anymore.

When considering the proof of $H \vdash \neg\nu\kappa.\phi$, we have again to distinguish two cases depending on whether or not an assumption involving N and $\nu\kappa.\phi$ belongs to H. If $N' : \nu\kappa.\phi \in H$, with $N \equiv N'$, a contradiction is reached and the derivation cannot be continued. Otherwise, to prove that N satisfies $\neg\nu\kappa.\phi$ under H, one has to establish that N satisfies $\neg\phi[\nu\kappa.\phi/\kappa]$ under the assumptions that are obtained from H by discharging all the formulae containing $\nu\kappa.\phi$ as a proper subformula.

State Formulae. Rules in Table 14 can be used to prove state formulae.

Table 14. Proof System: State Formulae

$$\dfrac{H \vdash N_2 : \phi\sigma}{H \vdash N_1 : \rho \to \phi}\; N_1 \ominus \xrightarrow[\sigma]{\rho} N_2 \qquad \dfrac{\left\{H \vdash N_2 : \neg\phi\sigma \;\middle|\; \exists N_2, \sigma. N_1 \ominus \xrightarrow[\sigma]{\rho} N_2\right\}}{H \vdash N_1 : \neg\rho \to \phi}$$

$$\dfrac{H \vdash N_2 : \phi}{H \vdash N_1 : \rho \leftarrow \phi}\; N_1 \oplus \xrightarrow{\rho} N_2 \qquad \dfrac{H \vdash N_2 : \neg\phi}{H \vdash N_1 : \neg@l \leftarrow \phi}\; N_1 \oplus \xrightarrow{\rho} N_2$$

To verify that a net N_1 satisfies $\rho \to \phi$ one has to prove that after consuming a resource specified by ρ, N_1 satisfies ϕ. Hence, a proof for $H \vdash N_1 : \rho \to \phi$ can be obtained by providing a net N_2, such that $N_1 \ominus \xrightarrow[\sigma]{\rho} N_2$, and a proof for $H \vdash N_2 : \phi\sigma$.

Conversely, a net satisfies $\neg(\rho \to \phi)$ if a resource described by ρ cannot be consumed without obtaining a net satisfying $\neg\phi$. For this reason, to prove that

N_1 satisfies $\neg(\rho \to \phi)$ under the assumptions H one has to prove that each N_2, such that $N_1 \ominus^{\rho}_{\sigma} N_2$, satisfies $\neg\phi\sigma$ under assumptions H.

For the production rules, if one can prove that the introduction of a resource specified by ρ leads to a net satisfying ϕ, then $\rho \leftarrow \phi$ is satisfied: If $N_1 \oplus^{\rho} N_2$, then a proof for $H \vdash N_1 : \rho \leftarrow \phi$ is obtained from a proof for $H \vdash N_2 : \phi$.

Finally, since a net satisfies $\neg\rho \leftarrow \phi$ if and only if it satisfies $\rho \leftarrow \neg\phi$, to prove that N_1 satisfies $\rho \leftarrow \phi$ under assumptions H, it is sufficient to exhibit a net N_2, such that $N_1 \oplus^{\rho} N_2$, and a proof for $H \vdash N_2 : \phi$.

Temporal Formulae. The rules of the proof system for temporal formulae are presented in Table 15. Some of them are reminiscent of standard rules for Hennessy-Milner Logic.

Table 15. Proof System: Temporal Formulae

$$\frac{H \vdash N_2 : \phi}{H \vdash N_1 : \langle\alpha\rangle\phi} \; N_1 \xrightarrow{\alpha} N_2 \qquad \frac{\left\{H \vdash N_1 : \neg\phi \mid N_1 \xrightarrow{\alpha} N_2\right\}}{H \vdash N_1 : \neg\langle\alpha\rangle\phi}$$

$$\frac{H \vdash N_2 : \phi}{H \vdash N_1 : \langle\star\rangle\phi} \; N_1 \xrightarrow{\lambda} N_2 \text{ and } fn(N) \cup fn(\phi) \Uparrow \lambda$$

$$\frac{\left\{H \vdash N_1 : \neg\phi \mid \forall N_1 \xrightarrow{\lambda} N_2 \text{ and } fn(N_1) \cup fn(\phi) \Uparrow \lambda\right\}}{H \vdash N_1 : \neg\langle\star\rangle\phi}$$

To prove $H \vdash N_1 : \langle\alpha\rangle\phi$ one has to provide a net N_2, such that $N_1 \xrightarrow{\alpha} N_2$, and then a proof for $H \vdash N_2 : \phi$. Conversely, to prove that N_1 satisfies $\neg\langle\alpha\rangle\phi$ under assumptions H, one has to prove that, for each N_2, such that $N_1 \xrightarrow{\alpha} N_2$, N_2 satisfies $\neg\phi$ under assumptions H.

Similarly, to prove that N_1 satisfies $\langle\star\rangle\phi$, one has to provide a transition label λ and a net N_2, such that $N_1 \xrightarrow{\lambda} N_2$, and a proof for $H \vdash N' : \phi$. However, when considering $\langle\star\rangle$, to avoid infinite branching, not all the transitions are taken into account. As discussed in Section 4.2, only transition labels that are interesting ($\Uparrow$) for $fn(N_1) \cup fn(\phi)$ are considered. All the ignored transitions can be obtained from the considered ones by renaming localities that do not appear in N_2 and in ϕ.

Finally, a proof for $H \vdash N_1 : \neg\langle\star\rangle\phi$ is obtained by proving that for each λ and N_2, such that $N_1 \xrightarrow{\lambda} N_2$ when λ is interesting for $fn(N_1) \cup fn(\phi)$, N_1 satisfies ϕ under assumptions H.

Nominal Formulae. In Table 16, the rules for nominal formulae are presented.

To prove $H \vdash N : \exists l.\phi$ one has to provide a locality l' and a proof for $H \vdash N : \phi[l'/l]$. Thanks to Lemma 1, which permits consistently replacing names

Table 16. Proof System: Nominal Formulae

$$\frac{H \vdash N : \phi[l'/l]}{H \vdash N : \exists l.\phi}\ l' \in fn(N) \cup fn(\phi) \cup \{sup(fn(N) \cup fn(\phi))\}$$

$$\frac{\{H \vdash N : \neg\phi[l'/l] | l' \in fn(N) \cup fn(\phi) \cup \{sup(fn(N) \cup fn(\phi))\}\}}{H \vdash N : \neg\exists l.\phi}$$

$$\frac{\{H \vdash N : \phi[l'/l] | l' = sup(fn(N) \cup fn(\phi))\}}{H \vdash N : \between l.\phi}$$

$$\frac{\{H \vdash N : \phi[l'/l] | l' = sup(fn(N) \cup fn(\phi))\}}{H \vdash N : \between l.\phi}$$

$$\frac{H \vdash N' : \phi}{H \vdash N : l \circledR \phi}\ N \equiv \nu\, l.N' \text{ and } reveal(l, N')$$

$$\frac{\{H \vdash N' : \neg\phi \mid N \equiv \nu\, l.N' \text{ and } reveal(l, N')\}}{H \vdash N : \neg l \circledR \phi}$$

in nets and formulae, it is sufficient to choose such locality among those belonging to $fn(N) \cup fn(\phi) \cup \{sup(fn(N) \cup fn(\phi))\}$. Similarly, a proof for $H \vdash N : \neg\exists l.\phi$ is obtained by proving $H \vdash N : \phi[l'/l]$, for each $l' \in fn(N) \cup fn(\phi) \cup \{sup(fn(N) \cup fn(\phi))\}$.

Rules for handling sequents involving fresh name quantifiers are similar. Indeed, a net N satisfies $\neg\between l.\phi$ if and only if N satisfies $\between l.\neg\phi$. If $l' = sup(fn(N) \cup fn(\phi))$, to prove $H \vdash N : \between l.\phi$ (resp. $H \vdash N : \neg\between l.\phi$), one has to prove that N satisfies $\phi[l'/l]$ (resp. $\neg\phi[l'/l]$).

Finally, if $N \equiv \nu\, l.N'$ and N' reveals l, then a proof for $H \vdash N : l \circledR \phi$ can be obtained from a proof for $H \vdash N' : \phi$. Conversely, to prove $H \vdash N : \neg l \circledR \phi$ one has to prove that each N' ($N \equiv \nu\, l.N'$) that reveals l satisfies $\neg\phi$.

Mobility Formulae. To prove formulae involving *mobility*, the rules in Table 17 have to be used. A net N_1 satisfies $\langle l_1 : \phi_1 \blacktriangleright l_2\rangle.\phi_2$ if a process satisfying ϕ_1 can be spawned from l_1 to l_2 contributing to form a net satisfying ϕ_2. Hence, when considering satisfaction of $\langle l_1 : \phi_1 \blacktriangleright l_2\rangle.\phi_2$ by a net N_1, under assumptions H, one has to exhibit a net N_2 and a process P, such that $N_1 \xrightarrow{\nu\,\widetilde{l}.l_1:P\blacktriangleright l_2} N_2$, for which $H \vdash l_2 :: P : \phi_1$ and $H \vdash \nu\,\widetilde{l}.N_2 \parallel l_2 :: P : \phi_2$ are provable. Please notice that, the spawned process is analysed as located at l_2 when checking satisfaction of ϕ_1. Moreover, as described in Section 3.5, all names $(\widetilde{l})$ shared by the remote evaluated process (P) and the rest of the system (N_2) are considered public when it is checked that $l_2 :: P$ satisfies ϕ_1.

Conversely, to prove that N_1 satisfies $\neg\langle l_1 : \phi_1 \blacktriangleright l_2\rangle.\phi_2$, one has to prove that a process satisfying ϕ_1 cannot be spawned from l_1 to l_2 without leading to a net satisfying $\neg\phi_2$. In other words, a proof for $H \vdash N_1 : \neg\langle l_1 : \phi_1 \blacktriangleright l_2\rangle.\phi_2$ is obtained by proving that if $N_1 \xrightarrow{\nu\,\widetilde{l}.l_1:P\blacktriangleright l_2} N_2$, then it holds that $H \vdash l_2 :: P : \phi_1$ and $H \vdash \nu\,\widetilde{l}.N_2 \parallel l_2 :: P : \phi_2$.

Table 17. Proof System: Mobility Formulae

$$\frac{H \vdash l_2 :: P : \phi_1 \quad H \vdash \nu\, l.(N_2 \parallel l_2 :: P) : \phi_2}{H \vdash N : \langle l_1 : \phi_1 \blacktriangleright l_2 \rangle.\phi_2} \; N_1 \xrightarrow{\nu\, l.l_1 : P \blacktriangleright l_2} N_2$$

$$\frac{\{H \vdash l_2 :: P : \phi_1, H \vdash \nu\, l.(N_2 \parallel l_2 :: P) : \neg\phi_2 | N_1 \xrightarrow{\nu\, \tilde{l}.l_1 : P \blacktriangleright l_2} N_2\}}{H \vdash N_1 : \neg\langle l_1 : \phi_1 \blacktriangleright l_2 \rangle.\phi_2}$$

5 Proving Properties of Mobile Systems

In this section we show how the proposed proof system can be used for assessing correctness of different print client implementations. The simplest implementation for a print client that satisfies all the formulae of example 3, is simply obtained by considering the lazy client:

$$PClient_1 :: \mathbf{nil}$$

If ϕ_1, ϕ_2 and ϕ_3 are the formulae of Example 3 we have that:

- $\vdash PClient_1 :: \mathbf{nil} : \phi_1$ is a successful sequent;
- $\vdash PClient_1 :: \mathbf{nil} : \phi_2$ and $\vdash PClient_1 :: \mathbf{nil} : \phi_3$ can be reduced, after some application of rule (ν), to successful sequents.

The actual derivations for ϕ_1 and for ϕ_2 are represented in Table 18[1] and Table 19 respectively.

Table 18. A derivation for the empty client (ϕ_2)

$$\dfrac{\dfrac{PClient_1 :: \mathbf{nil} : \phi_1 \vdash PClient_1 :: \mathbf{nil} : [l : PrintSlot \lhd PrintServer]\,\mathbf{T}}{PClient_1 :: \mathbf{nil} : \phi_1 \vdash PClient_1 :: \mathbf{nil} : [l : PrintSlot \lhd PrintServer]\,\mathbf{T} \vee [\star]\,\phi_1}\,[*]}{PClient_1 :: \mathbf{nil} : \phi_1 \vdash PClient_1 :: \mathbf{nil} : [l : (Print, l) \rhd PrintServer]\,\mathbf{F}}\,[*]$$

$$\dfrac{PClient_1 :: \mathbf{nil} : \phi_1 \vdash \begin{array}{r} PClient_1 :: \mathbf{nil} : [l : (Print, l) \rhd Prin`tServer]\,\mathbf{F} \\ \wedge([l : PrintSlot \lhd PrintServer]\,\mathbf{T} \\ \vee\,[\star]\,\phi_1) \end{array}}{\vdash \begin{array}{r} PClient_1 :: \mathbf{nil} : \nu\kappa.\,[l : (Print, l) \rhd PrintServer]\,\mathbf{F} \\ \wedge([l : PrintSlot \lhd PrintServer]\,\mathbf{T} \\ \vee\,[\star]\,\kappa) \end{array}}$$

The above amounts to saying that every net which never interacts with the print server, implements a *correct* print client.

[1] The proof in Table 18 is split in two parts where $[*]$ is the contact point.

Table 19. A derivation for the empty client (ϕ_3)

$$\dfrac{\dfrac{PClient_1 :: \mathbf{nil} : \phi_3 \vdash \quad PClient_1 :: \mathbf{nil} : \neg\langle l : Print, l \rhd PrintServer\rangle\mathbf{T}}{PClient_1 :: \mathbf{nil} : \phi_3 \vdash PClient_1 :: \mathbf{nil} : \neg\langle l : Print, l \rhd PrintServer\rangle\mathbf{T} \wedge \neg\langle\star\rangle\phi_3} \quad PClient_1 :: \mathbf{nil} : \neg\langle\star\rangle\phi_3}{\vdash PClient_1 :: \mathbf{nil} : \nu\kappa.\neg\langle l : Print, l \rhd PrintServer\rangle\mathbf{T} \wedge \neg\langle\star\rangle\kappa}$$

Let us now consider an incorrect specification for a client:

$$PClient_2 :: \mathbf{in}(!l)@l_1.\mathbf{in}(!x)@l_2.\mathbf{in}(l)@x, !l'.\mathbf{nil}$$

We have that it satisfies **ClientProp1** and **ClientProp2**, but does not satisfy **ClientProp3**. This is due to the fact that, *subtle* interactions with the environment can take place leading to an action that *violates* the security constraints describe with formula **ClientProp3**.

Indeed, the system above presents the following computation:

$$N_1 = PClient_2 :: \mathbf{in}(!l)@l_1.\mathbf{in}(!x)@l_2.\mathbf{in}(x, !l')@l.\mathbf{nil}$$
$$\xrightarrow{PClient_2 : PrintServer \lhd l_1}$$
$$N_2 = PClient_2 :: \mathbf{in}(!x)@l_2.\mathbf{in}(x, !l')@PrintServer.\mathbf{nil}$$
$$\xrightarrow{PClient_2 : Print \lhd l_2}$$
$$N_3 = PClient_2 :: \mathbf{in}(Print, !l')@PrintServer.\mathbf{nil}$$
$$\xrightarrow{PClient_2 : Print, l_0 \lhd PrintServer}$$
$$N_4 = PClinet_3 :: \mathbf{nil}$$

where first locality $PrintServer$ is retrieved from l_1 then value $Print$ is retrieved from l_2 and, finally, tuple $(Print, l_0)$ is retrieved from $PrintServer$. The proof for $\vdash N_1 : \neg\phi_3$ is presented in Table 20.

Table 20. A derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{N_1 : \phi_3, N_2 : \phi_3, N_3 : \phi_3 \vdash N_4 : \mathbf{T}}{N_1 : \phi_3, N_2 : \phi_3, N_3 : \phi_3 \vdash N_3 : \langle l : Print, l \lhd PrintServer\rangle\mathbf{T}}}{N_1 : \phi_3, N_2 : \phi_3, N_3 : \phi_3 \vdash N_3 : \neg\neg\langle l : Print, l \lhd PrintServer\rangle\mathbf{T}}}{N_1 : \phi_3, N_2 : \phi_3, N_3 : \phi_3 \vdash N_3 : \neg(\neg\langle l : Print, l \lhd PrintServer\rangle\mathbf{T} \wedge \neg\langle\star\rangle\neg\phi_3)}}{N_1 : \phi_3, N_2 : \phi_3 \vdash N_3 : \neg\phi_3}}{N_1 : \phi_3, N_2 : \phi_3 \vdash N_2 : \langle\star\rangle\neg\phi_3}}{N_1 : \phi_3, N_2 : \phi_3 \vdash N_2 : \neg\neg\langle\star\rangle\neg\phi_3}}{N_1 : \phi_3, N_2 : \phi_3 \vdash N_2 : \neg(\neg\langle l : Print, l \rhd PrintServer\rangle\mathbf{T} \wedge \neg\langle\star\rangle\neg\phi_3)}}{N_1 : \phi_3 \vdash N_2 : \neg\phi_3}}{N_1 : \phi_3 \vdash N_1 : \langle\star\rangle\neg\phi_3}}{N_1 : \phi_3 \vdash N_1 : \neg\neg\langle\star\rangle\neg\phi_3}}{N_1 : \phi_3 \vdash N_1 : \neg(\neg\langle l : Print, l \rhd PrintServer\rangle\mathbf{T} \wedge \neg\langle\star\rangle\neg\phi_3)}}{\vdash N_1 : \neg\phi_3}$$

A correct implementation for a client would be the following:

$$PClient_3 :: \mathbf{in}(PrintSlot)@PrintServer.\\ \mathbf{out}(Print, PClient)@PrintServer.\\ \mathbf{in}(PrintOk)@PClient.X$$

It models the following behaviour: a process located at *PClient* first retrieves tuple $\langle PrintSlot \rangle$ at *PrintServer*, then sends a print request and waits for a result. One can prove, using the proof system, that all the properties of Example 3 are satisfied by this.

6 Conclusions and Future Works

In this paper we have presented a temporal logic for specifying properties of mobile and distributed system. The logic is inspired by Hennessy-Milner Logic (HML) and the $\mu-$calculus, but has novel features that permit dealing with state properties and to describe the effect of data and processes movements over different sites of nets. The logic is equipped with a sound and complete proof system based on the tableaux approach.

It is possible to prove that the proposed proof system is sound and, under the assumption that the set of possible configurations of the considered system is finite, that it is also complete. That proofs will appear in the full version of the paper.

To have concrete examples of systems properties, we interpreted our logical formulae relatively in terms of μKLAIM, a simplified version of KLAIM. However, most of its operators are not specific of this language but can be used for any naming passing calculus with localities.

The main limitation of the proposed approach is that in order to establish system properties detailed descriptions of the whole systems under consideration are required. Obviously, this is a very strong assumption for wide area networks, because, very often only some components of the system are known; and one has only a limited knowledge of the overall context in which the component is operating. Nevertheless, one would like to guarantee that components well behave whenever the context guarantees specific resources or interactions.

For this reason we plan to set up a framework for specifying contexts for μKLAIM nets. By means of contexts, we will be able to provide abstract specifications of a given system and avoid describing all of its components in full. Indeed, some of these components could be known or implemented only at a later stage. Then, the implemented component can be removed from the context and added to the implemented part thus performing a *concretion* operation. We aim at setting up a framework that would guarantee preservation of satisfaction of formulae at each stage of refinement, if the introduced implementation *agrees* with the original specification.

Acknowledgements. We are grateful to the anonymous referees and to Diego Latella for many useful comments and suggestions that have helped in removing inaccuracies and improving the presentation.

References

1. L. Bettini, V. Bono, , R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The klaim project: Theory and practice. In *Global Computing*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer, 2003.
2. M. Bugliesi, G. Castagna, and S. Crafa. Reasoning about security in mobile ambients. In *Concur 2001*, number 2154 in Lecture Notes in Computer Science, pages 102–120. Springer, 2001.
3. L. Caires and L. Cardelli. A spatial logic for concurrency (part i). *Inf. Comput.*, 186(2):194–235, 2003.
4. L. Caires and L. Cardelli. A spatial logic for concurrency - ii. *Theor. Comput. Sci.*, 322(3):517–565, 2004.
5. L. Cardelli and A. D. Gordon. Mobile ambients. *Theor. Comput. Sci.*, 240(1):177–213, 2000.
6. L. Cardelli and A. D. Gordon. Ambient logic. *Mathematical Structures in Computer Science*, 2005. to appear.
7. N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(10):444–458, October 1989. Technical Correspondence.
8. G. Castagna, G. Ghelli, and F. Zappa Nardelli. Typing mobility in the seal calculus. In *Concur 2001*, number 2154 in Lecture Notes in Computer Science, pages 82–101. Springer, 2001.
9. G. Castagna and J. Vitek. Seal: A framework for secure mobile computations. In H. Bal, B. Belkhouche, and L. Cardelli, editors, *Internet Programming Languages*, number 1686 in Lecture Notes in Computer Science, pages 47–77. Springer, 1999.
10. M. Dam. Model checking mobile processes. *Journal of Information and Computation*, 129(1):35–51, 1996.
11. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
12. R. De Nicola and M. Loreti. A Modal Logic for Mobile Agents. *ACM Transactions on Computational Logic*, 5(1), 2004.
13. G. Ferrari, C. Montangero, L. Semini, and S. Semprini. Mark, a reasoning kit for mobility. *Automated Software Engineering*, 9:137–150, 2002.
14. C. Fournet, G. Gonthier, J. J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In U. Montanari and V. Sassone, editors, *Proc. of 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer-Verlag, 1996.
15. M. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *LICS*, pages 214–224, 1999.
16. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
17. D. Gelernter. Multiple Tuple Spaces in Linda. In J. G. Goos, editor, *Proceedings, PARLE '89*, volume 365 of *Lecture Notes in Computer Science*, pages 20–27, 1989.
18. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, Jan. 1985.
19. M. Hennessy and J. Riely. Distributed processes and location failures. *Theoretical Computer Science*, 266(1–2):693–735, Sept. 2001.
20. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173(1):82–120, 2002.

21. P. McCann and G.-C. Roman. Compositional programming abstraction for mobile computing. *IEEE Transactions on Software Engineering*, 24(2):97–110, 1998.
22. R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114:149–171, 1993.
23. R. Milner, J. Parrow, and J. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992.
24. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.
25. P. T. Wojciechowski and P. Sewell. Nomadic pict: Language and infrastructure design for mobile agents. In *ASA/MA*, pages 2–12. IEEE Computer Society, 1999.

Probabilistic Linda-Based Coordination Languages

Alessandra Di Pierro[1], Chris Hankin[2], and Herbert Wiklicky[2]

[1] Dipartimento di Informatica, Universitá di Pisa, Italy
[2] Department of Computing, Imperial College London, UK

Abstract. Coordination languages are intended to simplify the development of complex software systems by separating the coordination aspects of an application from its computational aspects. Coordination refers to the ways the independent active pieces of a program (e.g. a process, a task, a thread, etc.) communicate and synchronise with each other. We review various approaches to introducing probabilistic or stochastic features in coordination languages. The main objective of such a study is to develop a semantic basis for a quantitative analysis of systems of interconnected or interacting components, which allows us to address not only the functional (qualitative) aspects of a system behaviour but also its non-functional aspects, typically considered in the realm of performance modelling and evaluation.

1 Introduction

An early example of a Coordination Language was Linda [1]; Gelernter and Carriero offer the following equation [2]:

$$\text{Concurrent Programming} = \text{Computation} + \text{Coordination}$$

The intention being that Coordination Languages are "glue" languages for controlling the various computational components of a concurrent program.

Linda is an example of a *Shared Data Space* coordination language – the glue is provided by interaction through a shared tuple space. Alternative ways of synchronising components could be Message Passing, as in the Manifold model [3], or broadcast (cf Sands use of the CBS calculus [4]).

In this paper we consider various ways in which probabilities/quantities can be added to this basic paradigm; we distinguish between a data-driven and a schedule-driven approach. We also consider ways in which mobility can be added.

Our main objective is to develop a semantic basis for a quantitative analysis of networks. A quantitative analysis allows in general for the consideration of more "realistic" situations. For example, a probabilistic analysis allows for establishing the security of a system up to a given tolerance factor expressing how much the system is actually vulnerable. This is in contrast to a qualitative analysis which typically might be used to validate the absolute security of a given

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 120–140, 2005.
© Springer-Verlag Berlin Heidelberg 2005

system. In a distributed environment quantitative analysis is also of a great practical use in the consideration of timing issues which involve the asynchronous communications among processes running with different clocks.

The rest of this paper is structured into three main parts. First we consider how probabilities/quantities might be added to a Linda-like language. We consider a data-driven approach as presented in [5]; we also consider a schedule driven approach where probabilities/quantities are explicitly attached to parallel operators. We then consider a slightly more complicated language which includes mobility by introducing located processes and distributing the tuple space. For this part we will survey the approaches in [6,7,8]. We consider two alternatives: one in which nodes are visited with a certain probability (the discrete case) and the other in which nodes are visited at a certain rate (the continuous case). Finally, we consider an analysis framework for studying the properties of networks of processes; we briefly consider two variants which correspond to the discrete and continuous case respectively.

2 Linda

Linda [1] is a coordination language that relies on an asynchronous and associative communication mechanism based on a shared global space called the Tuple Space (TS), consisting of a multiset of tuples. The Linda language provides four simple operations for manipulating tuples by introducing, removing and reading tuples from the space TS. These operations allow processes to communicate and synchronise by interacting with the Tuple Space.

In order to investigate the introduction of quantitative information in the Linda paradigm, [5] introduces a minimal language, called Linda Calculus or LinCa, which includes the Linda core calculus expressed via only three constructs: prefix, parallel composition and replication. The syntax is presented in Table 1. **nil** represents the inactive process. The **out**(e) action causes a tuple to be deposited into the tuple space. The **in** t $(\boldsymbol{x})$ and **read** t $(\boldsymbol{x})$ actions both input tuples from the tuple space; the tuple is required to match the pattern t and the fields of the matching tuple are bound to the components of $\boldsymbol{x}$. The **in** action removes one matching tuple from the tuple space whereas **read** is non-destructive.

The original Linda language also includes an action **eval** which is similar to **out**(e), but capable of creating processes: for each tuple element which is a function, this primitive creates a new process to evaluate the function. Once all functions have been evaluated, **eval** will place the resulting tuple into the tuple space.

The semantics of LinCa is shown in Table 2. This is a small-step operational semantics. The configurations or states consist of a process and a tuple space, TS. The tuple space is essentially a multiset; $\oplus$ represents multiset union and $-$ represents multiset subtraction. We use $\triangleright$ to represent pattern matching; the exact details of how a tuple matches a pattern need not concern us in this paper. We have omitted the symmetric rule for parallel composition. Finally, we use the

Table 1. The LinCa syntax

$P ::=$	$\mathbf{nil}$	null process
$\mid$	$\mathbf{out}(e).P$	output prefix
$\mid$	$\mathbf{in}\ t\ (\boldsymbol{x}).P$	input prefix
$\mid$	$\mathbf{read}\ t\ (\boldsymbol{x}).P$	non-destructive input prefix
$\mid$	$P \mid P$	parallelism
$\mid$	$!\mathbf{in}\ t\ (\boldsymbol{x}).P$	replication

Table 2. LinCa semantics

$$[\mathbf{out}(e).P, TS] \longrightarrow [P, TS \oplus e]$$

$$\frac{\exists e \in TS : e \rhd t}{[\mathbf{in}\ t\ (\boldsymbol{x}).P, TS] \longrightarrow [P[e/\boldsymbol{x}], TS - e]}$$

$$\frac{\exists e \in TS : e \rhd t}{[\mathbf{read}\ t\ (\boldsymbol{x}).P, TS] \longrightarrow [P[e/\boldsymbol{x}], TS]}$$

$$\frac{[P, TS] \longrightarrow [P', TS']}{[P \mid Q, TS] \longrightarrow [P' \mid Q, TS']}$$

$$\frac{[\mathbf{in}\ t\ (\boldsymbol{x}).P, TS] \longrightarrow [P', TS']}{[!\mathbf{in}\ t\ (\boldsymbol{x}).p, TS] \longrightarrow [P' \mid !\mathbf{in}\ t\ (\boldsymbol{x}).P, TS]}$$

notation $P[e/\boldsymbol{x}]$ to represent the process P where all instances of the components of $\boldsymbol{x}$ (i.e. $x_1, \ldots, x_n$) have been replaced by corresponding "values" from e.

2.1 Adding Probabilities/Quantities

In the recent literature three proposals have been presented aimed at extending the basic coordination model à la Linda with quantities (probabilities, priorities, rates). These proposals follow two main approaches:

- Data Driven: In this approach the quantitative information is added to the data (tuples); it is adopted in [5] to define two quantitative versions of the core Linda language LinCa called PrioLinCa and ProbLinCa respectively (cf. Section 2.2). The main objective of [5] is the investigation of the expressive power of the different quantitative extensions compared to the basic paradigm.
- Schedule Driven: In this approach quantitative information is added to the "processes"; this is the approach taken in pKLAIM [6,7] and StocKLAIM

[8], where the motivation is more analysis-oriented. We will describe these two proposals in Section 3.2 and Section 3.3 in the context of a coordination language which extends Linda with distributed programming and mobility features.
We will also adopt this approach to define in Section 2.3 alternative versions of the prioritised and probabilistic LinCa introduced in [5].

2.2 Data Driven Approach

The starting point of the approach in [5] is the observation that nondeterminism is inherent in the definition of the Linda primitives. It occurs when a tuple becomes available on which more than one **in** t $(\boldsymbol{x})$ or **read** t $(\boldsymbol{x})$ action were suspended, or similarly when there is more than one tuple matching $\boldsymbol{x}$ in a **in** t $(\boldsymbol{x})$ or **read** t $(\boldsymbol{x})$ operation. This nondeterminism can be controlled by labelling tuples with quantities that can be interpreted respectively as priority or probability. For the resulting models, called PrioLinCa and ProbLinCa, Bravetti et al. have shown the following results:

- LinCa is not Turing complete (termination is decidable)
- PrioLinCa is Turing complete (encoding of RAM)
- ProbLinCa can solve the Leader Election problem; neither LinCa nor PrioLinCa can.

PrioLinCa. In PrioLinCa priorities (positive natural numbers) are added as attributes of tuples. The significant change in the language is in the semantics of **in** and **read**. For example the rule for **in** becomes:

$$\frac{\exists e \in TS : e \rhd t \;\; \forall e' \in TS : e' \rhd t \Rightarrow prio(e) \geq prio(e')}{[\mathbf{in}\ t\ (\boldsymbol{x}).P, TS] \longrightarrow [P[e/\boldsymbol{x}], TS - e],}$$

where $prio(e)$ denotes the quantity labelling the tuple e. A matching tuple is only removed from the tuple space if its priority is higher than any other matching tuple.

Rather than use priorities, it is possible to take a probabilistic approach which is exemplified by the alternative calculus called ProbLinCa.

ProbLinCa. In ProbLinCa weights (positive real numbers) are added as attributes of tuples. A tuple is then selected with a probability which is proportional to its weight. This is reflected in the semantics of the language by defining transition rules which probabilistically determine the next state according to a distribution which depend on the basic action of the starting state. Thus the rule for **in** becomes:

$$\frac{\exists e \in TS : e \rhd t}{[\mathbf{in}\ t\ (\boldsymbol{x}).P, TS] \longrightarrow \rho}$$

where ρ is a distribution on states which is computed as follows:

$$\rho(s) = \begin{cases} \dfrac{weight(e) \cdot TS(e)}{\sum_{e' \in TS : e' \rhd t} weight(e') \cdot TS(e')} & \text{if } s = [P[e/\boldsymbol{x}], TS - e], \\ & e \rhd t, e \in TS \\ 0 & \text{otherwise.} \end{cases}$$

where $TS(e)$ is the number of occurrences of the tuple e in TS. The probability of picking a particular tuple is thus computed in the following way:

- if the tuple, e, matches the pattern, t, the probability is the weight of the tuple times the number of occurrences of that tuple, normalised by the sum of the weights times multiplicities of all matching tuples.
- otherwise, when e doesn't match t, the probability is zero.

The rule for the **read** action determines an analogous distribution, while the rule for **out**(e) deterministically (i.e. with probability 1) leads to the state where the tuple e is added to the space independently of its weight:

$$[\mathbf{out}(e).P,\, TS] \longrightarrow \rho,$$

where $\rho([P,\, TS \oplus e]) = 1$ and $\rho(s) = 0$ for all the other states.

Finally, the rule for the parallel composition $P_1 \mid P_2$ nondeterministically chooses among the probability distributions determined by the transition rules for P_1 and P_2.

2.3 Schedule Driven Approach

In this section we propose alternative quantitative extensions of LinCa by adopting a schedule driven approach. In this approach we add priorities or probabilities to the operators – in particular, to the parallel operator.

A useful notion for the definition of a prioritised or probabilistic scheduler is the notion of "active state" which identifies those processes (essentially those prefixed by a **in** t $(\boldsymbol{x})$ or **read** t $(\boldsymbol{x})$ action) that are able to make a transition (essentially are not blocked awaiting for a tuple to become available).

Definition 1. *We define the set Active of* active *states as*

$$\begin{aligned} \mathit{Active} = \{[P,TS] \mid P \equiv \mathbf{in}\ t\ (\boldsymbol{x}).P' \ \mathit{and}\ \exists e \in TS : e \rhd t\} \\ \cup \{[P,TS] \mid P \equiv \mathbf{read}\ t\ (\boldsymbol{x}).P' \ \mathit{and}\ \exists e \in TS : e \rhd t\} \\ \cup \{[P,TS] \mid P \equiv \mathbf{out}(e).P'\}. \end{aligned}$$

Prioritised Scheduling. We replace the LinCa parallel composition $P \mid P$ by the prioritised parallel operator $p_1 : P_1 \mid p_2 : P_2$ where p_1 and p_2 are numbers (e.g. positive natural numbers as in PrioLinCa) expressing some priorities. A prioritised scheduler will (nondeterministically) select the state with higher priority among the active ones. Thus the semantics of the prioritised parallel operator can be defined by the rule:

$$\frac{[P_1,\, TS] \longrightarrow [P_1',\, TS'] \text{ and } p_1 \geq p_2}{[p_1 : P_1 \mid p_2 : P_2,\, TS] \longrightarrow [p_1 : P_1' \mid p_2 : P_2,\, TS']}$$

and the symmetric rule with P_2 in the premise.

From this semantics we can retrieve the data driven semantics of PrioLinCa. To see this, define the weight $\mathit{weight}(s)$ of a state $s = [P,TS]$ as the maximal weight of a matching tuple for P if s is active; $\mathit{weight}(s) = 0$ otherwise. Then assume that in the previous rule, priorities p_1 and p_2 are defined respectively as $\mathit{weight}([P_1,\, TS])$ and $\mathit{weight}([P_2,\, TS])$.

Table 3. Schedule Driven Probabilistic Semantics

$$[\mathbf{out}(e).P,\, TS] \longrightarrow_1 [P,\, TS \oplus e]$$

$$\frac{\exists e \in TS : e \rhd t}{[\mathbf{in}\ t\ (\boldsymbol{x}).P,\, TS] \longrightarrow_1 [P[e/\boldsymbol{x}],\, TS - e]}$$

$$\frac{\exists e \in TS : e \rhd t}{[\mathbf{read}\ t\ (\boldsymbol{x}).P,\, TS] \longrightarrow_1 [P[e/\boldsymbol{x}],\, TS]}$$

$$\frac{[P_i,\, TS] \longrightarrow_p [P_i',\, TS']}{[|_{j=1}^{n}\ p_j : P_j,\, TS] \longrightarrow_{p \cdot \tilde{p}_i} [p_i : P_i' \mid |_{j=1, j\neq i}^{n}\ p_j : P_j,\, TS']}$$

$$\frac{[P[e/\boldsymbol{x}],\, TS] \longrightarrow_p [P',\, TS']}{[A(e),\, TS] \longrightarrow_p [P',\, TS']} \qquad \text{if } A(\boldsymbol{x}) \equiv P$$

Probabilistic Scheduling. A probabilistic LinCa can be defined in the schedule-driven approach by replacing the parallel operator $P \mid P$ in the syntax of LinCa by a probabilistic one $p_1 : P_1 \mid p_2 : P_2$, where p_1 and p_2 are probabilities, that is real numbers in $[0, 1]$. Alternatively, we can let p_1 and p_2 range over the interval $[0, \infty)$: the normalisation process occurring at run-time guarantees that our quantities will indeed be transformed into probabilities.

The semantics of this alternative probabilistic Linda language can be defined in the usual SOS style via a probabilistic transition system $(S, \longrightarrow_p)$, where the the parameter p in the transition relation $\longrightarrow_p$ on states specifies the probability of a single step transition from one state to another. The rules defining $\longrightarrow_p$ are given in Table 3.

For the probabilistic parallel composition, in line with our previous work we opted for a more convenient n-ary version rather than the binary version used in Linda. In this rule the probability p_i is normalised to take account of the fact that the other branches of the parallel operator might be blocked. More precisely, we define the cumulative probability, $C_{[P,\,TS]}$, of all active processes in a parallel composition $P = |_{j=1}^{n}\ p_j : P_j$ as

$$C_{[P,\,TS]} = \sum_j \{p_j \mid [P_j,\, TS] \in \mathit{Active}\}.$$

Then the normalised probability $\tilde{p}_i$ is given by $\frac{p_i}{C_{[P,\,TS]}}$ if at least one of the two processes in P is active, and zero otherwise.

Replication introduces a new parallel operator; we must add probabilities to this:

$$\frac{[\mathbf{in}\ t\ (\boldsymbol{x}).P,\, TS] \longrightarrow_q [P',\, TS']}{[!\mathbf{in}\ t\ (\boldsymbol{x}).P,\, TS] \longrightarrow_q [p : P' \mid (1-p) :!\mathbf{in}\ t\ (\boldsymbol{x}).P,\, TS]}$$

This raises an issue about the choice of a value for p. This could be avoided by adding named processes and recursion rather than replication. We therefore introduce process *constants*, ranged over by A, and recursive definitions of the

form $A(\boldsymbol{x}) \equiv P$. The transition rule for recursion simply models the execution of a call to a procedure named A.

As a comparison with the probabilistic semantics for ProbLinCa defined in [5], we observe that the probabilistic model at the base of our semantics is *generative* according to the classification introduced in [9]: at each step the scheduler can select the next state according to one single probability distribution over the states. In the data driven semantics the probabilistic transition system conforms to the *reactive* model of probability instead: the scheduler can choose among different distributions depending on the (out/in/read) action taken. As a consequence our semantics contains strictly more information than the data driven semantics.

3 Distributed Tuple Spaces: KLAIM

The original Linda primitives are not completely adequate for programming distributed systems composed of mobile components. The KLAIM language (Kernel Language for Agents Interaction and mobility) was introduced in [10] as a distributed mobile version of Linda which extends the Linda interaction model by replacing the single shared tuple space with multiple distributed tuples spaces and allowing for explicit manipulation of localities and locality names.

3.1 A Core KLAIM Calculus

As before we will identify a simple core language where we consider only the basic constructs for prefixing, parallel composition and recursion. Moreover, we restrict to the actions **out**, **in** and **read** excluding the other KLAIM primitives, namely $\mathbf{eval}(P)@\ell$ which allows for a remote evaluation of the process argument[1], and $\mathbf{newloc}(u)$ which creates a new location accessible via the locality variable u. We also omit the consideration of allocation environments, that is partial functions used in the the full KLAIM language for the linking of symbolic names to physical addresses of nodes. The syntax of this minimal language, which we call cKLAIM, is given in Table 4.

The idea is to 'localise' processes P and their tuple spaces at some sites s and to construct networks N out of such nodes. We assume that locations are unique, i.e. only one process is attached to each location. The primitive actions **out**, **in** and **read** must now specify the local tuple space they refer to; this is done by introducing in their syntax the suffix "$@\ell$".

The operational semantics of cKLAIM is a restriction of the semantics of full KLAIM as presented in [10]. This is a two levels semantics: there are rules describing local transitions $P \xrightarrow{action} P'$ which are labelled with some (possible) action, and a global network semantics $N \Longrightarrow N'$ which specifies how a whole network evolves. The transition relation $\Longrightarrow$ is defined in terms of the local semantics $\longrightarrow$.

[1] This is different from the operation $\mathbf{eval}(t)$ in Linda whose argument is a tuple.

Table 4. cKLAIM Process and Network Syntax

P	::= **nil**	null process
	$\mid$ **out**$(e)@\ell.P$	output prefix
	$\mid$ **in** t $(\boldsymbol{x})@\ell.P$	input prefix
	$\mid$ **read** t $(\boldsymbol{x})@\ell.P$	non-destructive input prefix
	$\mid$ $P \mid P$	parallelism
	$\mid$ **!in** t $(\boldsymbol{x})@\ell.P$	replication
N	::= $s :: P$	node
	$\mid$ $N_1 \parallel N_2$	composition

Table 5. Discrete and Continuous Time Network Syntax

N	::= $s ::^q P$	node
	$\mid$ $N_1 \parallel N_2$	composition

N	::= $s ::^\lambda P$	node
	$\mid$ $N_1 \parallel N_2$	composition

3.2 Probabilistic KLAIM

Our main motivation for adding probabilities/quantities to coordination languages is to support quantitative analysis of distributed systems. The techniques that we have developed, based on Discrete or Continuous Time Markov Chains, provide a strong link between program analysis and recent advances in Performance Analysis [11]. A primary application of our work is in the study of quantitative aspects in Language Based Security, relative to e.g. denial of service, viruses, epidemiology, etc.

We will consider here only a probabilistic version of cKLAIM, and omit a prioritised one. Based on the two layered semantics of cKLAIM we will introduce probabilities both on the local and the global level. Locally we introduce probabilities into the parallel construct (scheduling information); globally, we introduce two different versions: in one we associate a probability with each node, while in the other we associate a rate. As a result we obtain two probabilistic extensions of KLAIM according to a discrete time and a continuous time Markov chain model respectively.

The only changes to the syntax are thus, as before, the introduction of a probabilistic parallel composition with scheduling probabilities p (in the discrete time model) or scheduling rates λ (in the continuous time model) for nodes. These changes are depicted in Table 5 with p and λ positive real numbers.

Local Semantics. Local transitions are labelled with an *action* label and are of the form:

$$[P, TS] \xrightarrow{action}_p [P', TS].$$

Table 6. The Local Structural Semantics

$$[\mathbf{out}(t)@\ell.P, TS] \xrightarrow{o(t)@\ell}_1 [P, TS]$$

$$[\mathbf{in}(t)@\ell.P, TS] \xrightarrow{i(t)@\ell}_1 [P, TS]$$

$$[\mathbf{read}(t)@\ell.P, TS] \xrightarrow{r(t)@\ell}_1 [P, TS]$$

$$\frac{[P_j, TS] \xrightarrow{\mu}_p [P'_j, TS']}{[|_{i=1}^n p_i : P_i, TS] \xrightarrow{\mu}_{p \cdot p_j} [\ |_{j \neq i=1}^n P_i \mid P'_j, TS']}$$

$$\frac{[P[e/x], TS] \xrightarrow{\mu}_p [P', TS']}{[A(e), TS] \xrightarrow{\mu}_p [P', TS']} \quad \text{with } A(x) \equiv P$$

This does not correspond to an actual change of the (local) configuration of a node but indicates the *possibility* of a local transition. It will be up to the global scheduler to activate such a potential update. In the local semantics we only consider how the process P changes, while the local tuple spaces TS remains the same and again it will be the global semantics to determine or not an actual update of TS.

The local semantics is defined in Table 6. As in the original semantics for KLAIM, we use the label *action* to describe the activities performed in the evolution; thus, for example $o(t)@\ell$ refers to the action of sending the tuple t in the tuple space specified by ℓ, and $r(t)@\ell$ is the action of consuming the tuple t in the tuple space specified by ℓ.

Global Semantics. The global semantics relies on the idea that state changes (transitions) do occur at certain points in time. In the discrete time case, every time step the scheduler selects one node to initiate an update of the whole network according to a (normalised) probability q. In the continuous time case jumps from one network state to another occur at rates specified by the scheduling rates λ. These rates determine an exponentially distributed time between transitions from one configuration of the network into another, according to a continuous time Markov chain model (cf. e.g. [12,13,14]).

According to Table 5 a probabilistic KLAIM network is either of the form $N \equiv \|_{i=1}^n s_i ::^{q_i} P_i$ or $N \equiv \|_{i=1}^n s_i ::^{\lambda_i} P_i$. We define a network configuration as a pair $[N, TS]$ with TS a global tuple space which is constructed out of the local tuple spaces TS_i, i.e. $TS = (TS_1, TS_2, \ldots, TS_n) = (TS_i)_{i=1}^n$. We will denote a network configuration $[N, TS] = [\|_{i=1}^n s_i ::^{x_i} P_i, (TS_i)_{i=1}^n]$ also as:

$$\|_{i=1}^n s_i ::^{x_i} [P_i, TS_i] = s_1 ::^{x_1} [P_1, TS_1] \parallel s_2 ::^{x_1} [P_2, TS_2] \parallel \ldots s_n ::^{x_i} [P_n, TS_n],$$

where $x_i = q_i$ or $x_i = \lambda_i$.

Discrete Time Version. The discrete time semantics of KLAIM networks is defined as a Discrete Time Markov Chain (DTMC) where the states are the network configurations; we will denote by $\mathcal{N}$ the set of all such configurations. A discrete time random process is a sequence $\{X_t\}_{t=1}^{\infty}$ of random variables, i.e. of functions $X_t : \Omega \to S$ from a probability space Ω into a state space S. We will restrict our presentation to finite[2] state spaces S and identify random variables $X(t) = X_t$ with their associated probability distributions $\mathbf{P}(X_t = s)$, i.e. the probability that the random variable X_i will be in state s. For finite state spaces S we can represent this probability distribution with a (column) vector which we will also denote by X_t.

A discrete time random process with initial distribution X_0 is called a discrete time Markov chain if the distribution for X_{t+1} only depends on the previous distribution X_t

$$\mathbf{P}(X_{t+1} = s_{t+1} \mid X_0 = s_0, \ldots, X_t = s_t) = \mathbf{P}(X_{t+1} = s_{t+1} \mid X_t = s_t).$$

This allows us to determine the distribution X_{t+1} via $X_{t+1} = X_t\mathbf{P}(t)$ where $\mathbf{P}(t)$ is a stochastic matrix, i.e. a matrix with row sums equal to one. For so called homogeneous DTMCs we have $\mathbf{P}(t) = \mathbf{P}$ for all t and:

$$X_t = X_{t-1}\mathbf{P} \quad \text{or} \quad X_t = X_{t-n}\mathbf{P}^n$$

In the case of the discrete time KLAIM networks we define their semantics as a (homogeneous) DTMC as follows: The state space S is the set of all possible network configurations $\mathcal{N}$. We can restrict ourselves to the set of network configurations $\mathcal{N}(N(0))$ which are reachable from the initial configuration $N(0) = [N_0, TS_0]$ and assume that $\mathcal{N}(N(0))$ is finite. The entries in the transition matrix $\mathbf{P}$ are then defined by using the rules in Table 7 as:

$$\mathbf{P}_{N_i,N_j} = \begin{cases} \sum p_{ij} & \text{with } N_i \overset{p_{ij}}{\Longrightarrow} N_j \\ 0 & \text{otherwise.} \end{cases}$$

The rules in Table 7 describe how a global scheduler can utilise potential local transitions in order to update the global network configuration. The update is triggered by one of the nodes, at s_1, with a probability corresponding to its scheduling probability p_i. Each update involves at most two nodes at sites s_1 and s_2 in the context of the remaining nodes of the network, denoted by N. If $s_1 = s_2$ the rules in Table 7 have to be applied in the obvious way.

As some nodes could be blocked — e.g. because no matching tuple is available for an **in** to proceed — we have to use the normalised probabilities $\tilde{p}_i$. For this we define the set of *active* sites in a global configuration $[P, TS] = \|_{i=1}^{n} s_i ::^{p_i} [P_i, TS_i]$ as:

$$\mathit{Active}([P, TS]) = \{s_i \mid [P_i, TS_i] \xrightarrow{a}_p [P_i', TS_i']\}\}$$

[2] For countable infinite state spaces we have to use probability measures instead of probability distributions.

Table 7. Discrete Time Network Semantics

$$\frac{[P_1, TS_1] \xrightarrow{o(t)@s_2}_p [P_1', TS_1]}{s_1 ::^{p_1} [P_1, TS_1] \| s_2 ::^{p_2} [P_2, TS_2] \| N \overset{\tilde{p}\tilde{p}_1}{\Longrightarrow} s_1 ::^{p_1} [P_1', TS_1] \| s_2 ::^{p_2} [P_2, TS_2 \oplus t] \| N}$$

$$\frac{[P_1, TS_1] \xrightarrow{i(t)@s_2}_p [P_1', TS_1] \quad \exists e \in TS_2 : e \rhd t}{s_1 ::^{p_1} [P_1, TS_1] \| s_2 ::^{p_2} [P_2, TS_2] \| N \overset{\tilde{p}\tilde{p}_1}{\Longrightarrow} s_1 ::^{p_1} [P_1'[x/e], TS_1] \| s_2 ::^{p_2} [P_2, TS_2 - e] \| N}$$

$$\frac{[P_1, TS_1] \xrightarrow{r(t)@s_2}_p [P_1', TS_1] \quad \exists e \in TS_2 : e \rhd t}{s_1 ::^{p_1} [P_1, TS_1] \| s_2 ::^{p_2} [P_2, TS_2] \| N \overset{\tilde{p}\tilde{p}_1}{\Longrightarrow} s_1 ::^{p_1} [P_1'[x/e], TS_1] \| s_2 ::^{p_2} [P_2, TS_2] \| N}$$

i.e. a site s_i is active if its process P_i can make at least one local transition with some action a and probability p. Then we define

$$\tilde{p}_i = \frac{p_i}{C_{[P,TS]}} \quad \text{with} \quad C_{[P,TS]} = \sum_j \{p_j \mid s_j \in \mathit{Active}([P, TS])\}.$$

In a similar way we also have to normalise the local probability p. The active actions of a local configuration (in some network context) are given by:

$$\begin{aligned} \mathit{Active}([P_i, TS_i]) = \{&o(t)@s_j \mid [P_i, TS_i] \xrightarrow{o(t)@s_j}_p [P_i', TS_i']\} \\ \cup \{&i(e)@s_j \mid [P_i, TS_i] \xrightarrow{i(t)@s_j}_p [P_i', TS_i'] \text{ and } \exists e \in TS_j : e \rhd t\} \\ \cup \{&r(e)@s_j \mid [P_i, TS_i] \xrightarrow{r(t)@s_j}_p [P_i', TS_i'] \text{ and } \exists e \in TS_j : e \rhd t\} \end{aligned}$$

and with this we get the normalised local transition probabilities as:

$$\tilde{p} = \frac{p_i}{C_{[P,TS]}} \quad \text{with} \quad C_{[P_i,TS_i]} = \sum_j \{p_j \mid a_j \in \mathit{Active}([P_i, TS_i])\}.$$

If no node is active for a network configuration N we will force a diagonal entry $\mathbf{P}_{N,N} = 1$ to guarantee that $\mathbf{P}$ is indeed a stochastic matrix. Operationally this corresponds to introducing a self-transition or loop for stuck network configurations.

Continuous Time Version. In this model each node can initiate a network update independently at any time with a certain probability which is proportional to its rate. This parameter is specified by the superscript λ in the syntax of a node. We assume that these rates are independent from the time and therefore each node "fires", i.e. initiates an update, via a so called Poisson process (see e.g. [13–Sect 2.4]).

We model the continuous time semantics of KLAIM networks as a Continuous Time Markov Chains (CTMC), i.e. as a particular continuous time random process $\{X_t\}_{t\in[0,\infty)}$. Like in the case of DTMCs the dependency between the random variables $X_t = X(t)$ in a CTMC is very restricted: it depends only on (any) single previous moment. This means that there exist stochastic matrices $\mathbf{P}(t)$, with $t \in [0,\infty)$, such that we can compute the distribution X_t as:

$$X_t = X_{t-\triangle t}\mathbf{P}(\triangle t).$$

The matrices $\mathbf{P}(t)$ form a semi-group, i.e. $\mathbf{P}(0) = \mathbf{I}$ the identity matrix ($p_{ij}(0) = 1$ for $i = j$ and $p_{ij}(0) = 0$ otherwise) and for any $t, s \in [0,\infty)$ we have the so called semi-group property: $\mathbf{P}(s+t) = \mathbf{P}(s)\mathbf{P}(t)$.

It is possible to obtain the $\mathbf{P}(t)$ matrices as solutions to certain linear differential equations (cf. e.g. [13]). This allows us to specify the $\mathbf{P}(t)$s via the parameters describing these differential equations. These parameters are referred to as the rate or Q-matrix $\mathbf{Q} = (q_{ij})_{ij}$ which has the following properties:

- $0 \leq -q_{ii} < \infty$ for all i,
- $q_{ij} \geq 0$ for all $i \neq j$,
- $\sum_j q_{ij} = 0$ for all i.

From the Q-matrix of a system we can obtain the transition probabilities $\mathbf{P}(t)$ via:

$$\mathbf{P}(t) = \exp(t\mathbf{Q}) = \sum_{n=0}^{\infty} \frac{(t\mathbf{Q})^n}{n!}$$

In the case of the continuous time semantics for KLAIM networks we only need to specify the rate matrix $\mathbf{Q}$ using the rules in Table 8:

$$\mathbf{Q}_{N_i,N_j} = \begin{cases} \sum w_{ij} & \text{for } N_i \overset{w_{ij}}{\Longrightarrow} N_j \\ -\sum_{j\neq i} w_{ij} & \text{for } N_i = N_j \\ 0 & \text{otherwise.} \end{cases}$$

In Table 8 the relation $\overset{w_{ij}}{\Longrightarrow}$ between two network configurations N_i and N_j is labelled by rates w_{ij} which are obtained as a product between the firing rate λ_k of the node which initiates the update and the normalised probabilities $\tilde{p}$ of the local transitions occurring in the nodes involved in the update. The normalisation of the local transition probabilities is again needed in order to accommodate locally blocked transitions. The rates λ need no normalisation and we need no special treatment of completely blocked network configurations (this is achieved via the construction of the diagonal elements in $\mathbf{Q}$ and related to the basic fact that $\exp(0) = 1$).

The continuous time model realises true concurrency as several transitions *seem* to happen in "parallel". In fact, two transitions are actually never happening at exactly the same moment, as the probability for this is zero. However, after a single time unit we can observe that two or more transitions have happened.

Table 8. Continuous Time Network Semantics

$$\frac{[P_1, TS_1] \xrightarrow{o(t)@s_2}_p [P_1', TS_1]}{s_1 ::^{\lambda_1} [P_1, TS_1] \| s_2 ::^{\lambda_2} [P_2, TS_2] \| N \overset{\tilde{p}\lambda_1}{\Longrightarrow} s_1 ::^{\lambda_1} [P_1', TS_1] \| s_2 ::^{\lambda_2} [P_2, TS_2 \oplus t] \| N}$$

$$\frac{[P_1, TS_1] \xrightarrow{i(t)@s_2}_p [P_1', TS_1] \quad \exists e \in TS_2 : e \rhd t}{s_1 ::^{\lambda_1} [P_1, TS_1] \| s_2 ::^{\lambda_2} [P_2, TS_2] \| N \overset{\tilde{p}\lambda_1}{\Longrightarrow} s_1 ::^{\lambda_1} [P_1'[x/e], TS_1] \| s_2 ::^{\lambda_2} [P_2, TS_2 - e] \| N}$$

$$\frac{[P_1, TS_1] \xrightarrow{r(t)@s_2}_p [P_1', TS_1] \quad \exists e \in TS_2 : e \rhd t}{s_1 ::^{\lambda_1} [P_1, TS_1] \| s_2 ::^{\lambda_2} [P_2, TS_2] \| N \overset{\tilde{p}\lambda_1}{\Longrightarrow} s_1 ::^{\lambda_1} [P_1'[x/e], TS_1] \| s_2 ::^{\lambda_2} [P_2, TS_2] \| N}$$

This allows us to avoid considering "clashes" like for example two $\mathbf{in}(t)$ actions trying to access the same token: the probability of this happening vanishes. We can however ask for the probability that either of the two **in**'s is executed first and in this way determine the chances that the token in question has been consumed by the first or the second **in** after a given time (or, as also could be the case, that neither of them has already consumed the token).

3.3 Stochastic KLAIM

An alternative stochastic version of KLAIM is the proposal in [8]. The language defined in this work, called StocKLAIM, extends a core subset of KLAIM by associating to each action some specific rates representing the time taken by the action to be executed. Similarly to the continuous time version of our pKLAIM, this time is determined by random variables which are exponentially distributed, so that the operational semantics of the language can be represented in terms of stochastic processes and in particular as a Continuous Time Markov Chain.

In StocKLAIM the only source of probabilistic information is therefore the action prefix process whose syntax $(a, r).P$ allows for the specification of a rate by instantiating the name r. All the other constructs of the language, namely the choice $P + P$ and the parallel composition $P \mid P$ as well the network parallel operator $N \parallel N$ keep their nondeterministic syntax, so that for example no information (coming e.g. from statistical or other form of data which are available for a given application) can be specified at this level.

The straightforward relation of the structural operational semantics defined via a labelled transition system with the CTMC model, makes various tools and techniques which have been developed for stochastic model checking directly available for the analysis of a network specified in StochKLAIM. In fact one main motivation of this approach is towards the definition of logics and semantics based tools for the performance modelling and analysis of mobile and distributed application.

Alternatively, a probabilistic model like the one offered by probabilistic KLAIM (cf. Section 3.2) is more oriented (even when it is based on a CMTC model) towards a quantitative analysis of networks based on tools and techniques which are typical of program analysis; these are obtained by transforming the approaches that have been developed for a purely qualitative static analysis of program properties into probabilistic/quantitative ones suitable for a distributed setting (cf. Section 4).

4 Analysis

Given a program in probabilistic Linda or a network in probabilistic KLAIM we are interested in analysing properties such as the chances that a program terminates, or the chance that a token (worm) moves from one node in the network to another one in a given number of steps (or time interval). As in classical program analysis, also probabilistic properties can be expressed as solutions to a set of (in)equations. Since most properties are undecidable, an exact solution of these (in)equations is often not possible. We are thus led to construct reasonable approximations. Depending on the structure of the domain of the (in)equations there are various options regarding a formal definition of "reasonable approximation". In particular, we can consider the following two settings:

Order Theoretic: This is based on partial orders or lattice structures and aims at computing the best "safe" solution. Classical Abstract Interpretation [15,16] utilises the notion of a Galois Connection to achieve this aim.

Linear Structures: This is based on linear spaces or operator algebras and allows for the construction of least squares solution as "closest" fit. Probabilistic Abstract Interpretation [17] uses the so-called Moore-Penrose Pseudo-inverse for this purpose.

In this paper we will assume the second setting as the base of our treatment.

4.1 Probabilistic Abstract Interpretation

Probabilistic Abstract Interpretation (PAI) is a general framework introduced in [17] for the static analysis of probabilistic programs. Similarly to the classical abstract interpretation framework, it provides general techniques for constructing approximations of the semantics of a system relatively to a given property of interest. However, the *correctness* of these approximations which is guaranteed in the classical case by the *order-theoretic* notion of *Galois connection*, is replaced in PAI by a notion of *closeness* which includes some quantitative measurement of the loss of precision. This is obtained by moving from the order-theoretic setting of classical abstract interpretation to one based on linear spaces and linear operators, where the notion of so-called *Moore-Penrose pseudo-inverse* (see below) replaces the classical notion of a Galois connection. Moreover, the properties of the Moore Penrose inverse guarantees the *optimality* of the approximations constructed via PAI: they are the closest possible to the concrete semantics of the given system.

The definition of a probabilistic abstract interpretation is given in terms of *probabilistic domains*. A probabilistic domain is essentially a space which represents the distributions $Dist(S)$ on a state space S. In the general case including infinite dimensional vector spaces, a probabilistic domain is defined as the Hilbert space $\mathcal{H}(S) = \ell^2(S)$ on S (cf. [18,19]). However, in the finite dimensional case this is equivalent to consider the simple vector space $\mathcal{V}(S)$, built out of all linear combinations of elements from S with coefficients in $\mathbb{R}$:

$$\mathcal{V}(S) = \left\{ \sum c_s \boldsymbol{s} \mid c_s \in \mathbb{R},\ s \in S \right\}.$$

For the purpose of this work it is sufficient for us to consider only finite dimensional vector spaces, so we will present the PAI framework in this restricted setting.

Definition 2. *Let $\mathcal{C}$ an $\mathcal{D}$ be two probabilistic domains. A* probabilistic abstract interpretation *is a pair of bounded linear operators $\mathbf{A} : \mathcal{C} \to \mathcal{D}$ and $\mathbf{G} : \mathcal{D} \to \mathcal{C}$, between (the concrete domain) $\mathcal{C}$ and (the abstract domain) $\mathcal{D}$, such that $\mathbf{G}$ is the Moore-Penrose pseudo-inverse of $\mathbf{A}$, and vice versa.*

A particular PAI technique similar to the classical abstract interpretation technique defining a so-called *induced abstract semantics* consists in the following: Given a linear operator Φ on some Hilbert space $\mathcal{V}$ expressing the probabilistic semantics of a concrete system, and a linear abstraction function $\mathbf{A} : \mathcal{V} \mapsto \mathcal{W}$ from the concrete domain into an abstract domain $\mathcal{W}$, we compute the Moore-Penrose pseudo-inverse $\mathbf{G} = \mathbf{A}^\dagger$ of $\mathbf{A}$. The abstract semantics can then be defined as the linear operator on the abstract domain $\mathcal{W}$:

$$\Psi = \mathbf{A} \circ \Phi \circ \mathbf{G}.$$

Moore-Penrose Pseudo-Inverse. We can define the notion of a Moore-Penrose pseudo-inverse of a bounded linear operator $\mathbf{A} \in \mathcal{B}(\mathcal{H})$ on a Hilbert space $\mathcal{H}$ purely algebraically (cf. Section 4.7 of [20], and Section 8.43 of [21]). This is sufficient for the finite-dimensional setting, while for dealing with the infinite-dimensional case we need some topological considerations [22].

Definition 3. *Let $\mathcal{C}$ and $\mathcal{D}$ be two Hilbert spaces and $\mathbf{A} : \mathcal{C} \mapsto \mathcal{D}$ a bounded linear map between them. A bounded linear map $\mathbf{A}^\dagger = \mathbf{G} : \mathcal{D} \mapsto \mathcal{C}$ is the* Moore-Penrose pseudo-inverse *of $\mathbf{A}$ iff*

(i) $\mathbf{A} \circ \mathbf{G} = \mathbf{P}_A$, *and*
(ii) $\mathbf{G} \circ \mathbf{A} = \mathbf{P}_G$,

where $\mathbf{P}_A$ and $\mathbf{P}_G$ denote orthogonal projections onto the ranges of $\mathbf{A}$ and $\mathbf{G}$.

In the finite dimensional case, the Moore-Penrose pseudo inverse of a linear operator always exists and is unique; moreover various algorithms are known for its construction [23]. A general technique for computing the Moore-Penrose pseudo-inverse of infinite operators is to approximate them by a sequence of

finite dimensional operators. For infinite dimensional operators — i.e. operator algebras over infinite dimensional Hilbert spaces — various results guarantee the existence of the Moore-Penrose pseudo-inverse for every operator. For the general case we mention here the one in [20] (Theorem 4.24) which also states how one can "construct" the Moore-Penrose Pseudo-Inverse.

Probabilistic Abstract Interpretation and Classification. In many cases the abstraction is a surjective function. An alternative view of abstraction in this case is that it maps concrete values to equivalence classes. Equivalence relations can be represented by a particular kind of operators, namely classification operators. In the finite dimensional setting these can be described as follows.

We call an $n \times m$-matrix $\mathbf{K}$ a *classification* matrix, if it is a 0/1-matrix, where every row has exactly one non-zero entry and columns have at least one non-zero entry. Classification matrices are thus particular kinds of stochastic matrices. We denote by $\mathcal{K}(n,m)$ the set of all $n \times m$-classification matrices. Let $X = \{x_1, \ldots, x_n\}$ be a finite set. Then for each equivalence relation $\approx$ on X with $|X/_{\approx}| = m$, there exists a classification matrix $\mathbf{K} \in \mathcal{K}(n,m)$ and vice versa.

The Moore-Penrose pseudo-inverse of a classification matrix $\mathbf{K} \in \mathcal{K}(n,m)$ corresponds to its normalised transpose or adjoint (these coincide for real $\mathbf{K}$), i.e.

$$\mathbf{K}^{\dagger} = \mathcal{N}(\mathbf{K}^T) = \mathcal{N}(\mathbf{K}^*).$$

where the normalisation operation $\mathcal{N}$ is defined for a matrix $\mathbf{A}$ by:

$$\mathcal{N}(\mathbf{A})_{ij} = \begin{cases} \frac{\mathbf{A}_{ij}}{a_j} \text{ if } a_j = \sum_i \mathbf{A}_{ij} \neq 0 \\ 0 \quad \text{otherwise.} \end{cases}$$

4.2 Analysis – Discrete Case

In order to exploit the framework of Probabilistic Abstract Interpretation in the case of probabilistic KLAIM we need a semantics given in terms of linear (transition) operators. This is provided by the operators $\mathbf{P}$ and $\mathbf{P}(t)$ introduced in Section 3.2.

In the case of the discrete time model we have to consider a single step operator $\mathbf{P}$ which describes the probabilistic transitions between (network) configurations. For KLAIM we can construct this operator either as a direct encoding of the operational semantics (as in Section 3.2) or compositionally reflecting the two-layered semantics:

The local semantics defines a Probabilistic Transition System (PTS) — this is represented as a linear operator, in particular a stochastic matrix [24].

The global semantics is then constructed compositionally as the tensor product of the local semantics [25].

Based on the concrete semantics given by $\mathbf{P}$ we can construct an abstract induced semantics **GPA** using some abstraction operator $\mathbf{A}$ and its Moore-Penrose pseudo inverse $\mathbf{G} = \mathbf{A}^{\dagger}$. This amounts effectively to a "simplification" of the DTMC by reducing the dimension of the transition matrix $\mathbf{P}$.

Example 1. Consider the following 5×5 transition matrix:

$$\mathbf{P} = \begin{pmatrix} 0 & \frac{3}{4} & \frac{1}{4} & 0 & 0 \\ \frac{3}{4} & 0 & \frac{1}{4} & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{4} \\ 0 & 0 & \frac{1}{4} & 0 & \frac{3}{4} \\ 0 & 0 & \frac{1}{4} & \frac{3}{4} & 0 \end{pmatrix}$$

Suppose that we abstract states into one of two classes. This corresponds to partition the set of states in two equivalence classes which can suitably be represented via the classification operator $\mathbf{K}$ and its pseudo-inverse $\mathbf{K}^\dagger$:

$$\mathbf{K} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \text{ and } \mathbf{K}^\dagger = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{pmatrix}$$

In this case the abstract 2×2 transition matrix is:

$$\mathbf{K}^\dagger \mathbf{P} \mathbf{K} = \begin{pmatrix} \frac{3}{4} & \frac{1}{4} \\ \frac{1}{6} & \frac{5}{6} \end{pmatrix},$$

which can then be safely used to replace P to simplify our analysis. Note that in our PAI framework "safely" means that the approximation error we make is controllable, that is always quantifiable.

In fact, one advantage of the use of linear operators is that we can measure them. The standard way to measure the "size" of a linear operator is via an operator norm which in turn may have its origins in a vector norm. This allows us, for example, to quantify the error introduced by the abstraction. The close relation between least squares approximation and the Moore-Penrose pseudo-inverse (cf. e.g. [23]) guarantees that the abstract induced semantics is giving the closest (with respect to the Euclidean norm) approximation to the concrete behaviour among all the possible ones.

4.3 Analysis – Continuous Case

We can also apply this simplification technique to CTMCs. Concretely, we have to construct the generator $\mathbf{Q}$ as described in Section 3.2. The probability that the network configuration $N(t)$ at any time t is N_j starting from initial configuration N_i is then $\mathbf{P}(t)_{ij} = \mathbf{P}(N(t) = n_j \mid N(0) = N_i) = (\exp(t\mathbf{Q}))_{ij}$.

In the same way as with the discrete time model we can simplify the operator $\mathbf{P}(t)$ by subjecting it to an abstract interpretation.

In fact, the common method for effectively computing $\mathbf{P}(t)$ is closely related to a particular form of probabilistic abstract interpretation. It is based on the fact that every matrix — in particular the generator matrix $\mathbf{Q}$ — can be "abstracted" into a so called Jordan canonical form, e.g. [26–Chap 7] or [27–Sect III.12].

A *Jordan matrix* $\mathbf{J}$ is a square matrix of the form $\mathbf{J} = \mathrm{diag}(\mathbf{J}_{r_1}(\lambda_1), \ldots, \mathbf{J}_{r_m}(\lambda_m))$ with $\mathbf{J}_{r_i}(\lambda_i)$ so called *Jordan blocks*, i.e. $r_i \times r_i$ matrices of the form:

$$\mathbf{J}_{r_i}(\lambda_i) = \begin{pmatrix} \lambda_i & 1 & 0 & \cdots & 0 & 0 \\ 0 & \lambda_i & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_i & 1 \\ 0 & 0 & 0 & \cdots & 0 & \lambda_i \end{pmatrix} = \begin{pmatrix} \lambda_i & 0 & 0 & \cdots & 0 & 0 \\ 0 & \lambda_i & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_i & 0 \\ 0 & 0 & 0 & \cdots & 0 & \lambda_i \end{pmatrix} + \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix}$$

Theorem 1. *[27–Thm III.12.2] Any complex square matrix* $\mathbf{A}$ *is* similar *to a Jordan matrix* $\mathbf{J}$*, i.e. there exists an invertible matrix* $\mathbf{X}$ *such that* $\mathbf{A} = \mathbf{X}^{-1}\mathbf{J}\mathbf{X} = \mathbf{X}^{\dagger}\mathbf{J}\mathbf{X}$.

As every Jordan block $\mathbf{J}_{r_i}(\lambda_i)$ is the sum of a diagonal $\mathbf{D}_{r_i}(\lambda_i) = \mathrm{diag}(\lambda_i, \ldots, \lambda_i)$ and a strict upper triangular matrix $\mathbf{N}_{r_i}$ the same holds for Jordan matrices, i.e. $\mathbf{J} = \mathbf{D} + \mathbf{N}$. Furthermore, it is easy to see that $\mathbf{N}$ is *nilpotent*, i.e. there exists a $n \in \mathbb{N}$ such that $\mathbf{N}^m$ is the null matrix, and that $\mathbf{D}$ and $\mathbf{N}$ commute: $\mathbf{D}\mathbf{N} = \mathbf{N}\mathbf{D}$, see e.g. [26,27].

This allows for an efficient way of computing $\mathbf{P}(t) = \exp(t\mathbf{Q})$: With the Jordan canonical form $\mathbf{J}$ of $\mathbf{Q}$, i.e. $\mathbf{Q} = \mathbf{X}^{-1}\mathbf{J}\mathbf{X}$, we get:

$$\begin{aligned} \exp(t\mathbf{Q}) &= \sum_{k=0}^{\infty} \frac{1}{k!}(t\mathbf{Q})^k = \sum_{k=0}^{\infty} \frac{t^k}{k!}(\mathbf{X}^{-1}\mathbf{J}\mathbf{X})^k = \sum_{k=0}^{\infty} \frac{t^k}{k!}\mathbf{X}^{-1}\mathbf{J}^k\mathbf{X} \\ &= \mathbf{X}^{-1}\left(\sum_{k=0}^{\infty} \frac{1}{k!}(t\mathbf{J})^k\right)\mathbf{X} = \mathbf{X}^{-1}\exp(t\mathbf{J})\mathbf{X} \end{aligned}$$

Furthermore, we can compute the exponential of a Jordan matrix $\mathbf{J}$ easily (exploiting the fact that $\mathbf{D}$ and $\mathbf{J}$ commute):

$$\exp(t\mathbf{Q}) = \exp(t\mathbf{D} + t\mathbf{N})) = \exp(t\mathbf{D})\exp(t\mathbf{N}).$$

Finally, we observe that the exponential of diagonal matrices $\mathbf{D}$ is simply:

$$\exp(\mathrm{diag}(d_1, d_2, \ldots, d_n)) = \mathrm{diag}(\exp(d_1), \exp(d_2), \ldots, \exp(d_n))$$

and that for nilpotent matrices $\mathbf{N}$ the series

$$\exp(\mathbf{N}) = \sum_{k=0}^{\infty} \frac{1}{k!}\mathbf{N}^k = \sum_{k=0}^{m} \frac{1}{k!}\mathbf{N}^k$$

degenerates to a finite sum, with m the nilpotency of $\mathbf{N}$. In the case of diagonalisable matrices $\mathbf{Q}$ the Jordan canonical form is a diagonal matrix, i.e. $\mathbf{N}$ is the null matrix, which means that we obtain $\mathbf{P}(t) = \exp(t\mathbf{Q}) = \mathbf{X}^{-1}\exp(t\mathbf{D})\mathbf{X} = \mathbf{X}^{-1}\mathrm{diag}(\exp(td_1), \exp(td_2), \ldots, \exp(td_n))\mathbf{X}$.

This approach is a special instance of a general way for solving linear differential equations, see e.g. [28]. Unfortunately, the key property $\exp(\mathbf{X}^{-1}\mathbf{J}\mathbf{X}) = \mathbf{X}^{-1}\exp(\mathbf{J})\mathbf{X}$ does not hold if we consider proper abstractions, i.e. $\mathbf{X}^{\dagger}$ instead of $\mathbf{X}^{-1}$, i.e. $\exp(\mathbf{X}^{\dagger}\mathbf{J}\mathbf{X}) \neq \mathbf{X}^{\dagger}\exp(\mathbf{J})\mathbf{X}$. This is due to the fact that $(\mathbf{X}^{\dagger}\mathbf{J}\mathbf{X})^k \neq \mathbf{X}^{\dagger}\mathbf{J}^k\mathbf{X}$ as $\mathbf{X}\mathbf{X}^{\dagger} \neq \mathbf{I}$. The factor $\mathbf{X}\mathbf{X}^{\dagger}$ describes exactly the approximation error we introduce by considering the abstract in place of the concrete semantics.

5 Conclusions

We have explored the design space for adding quantitative information to coordination languages. We have used a basic Linda calculus for this. We showed that one could either add priorities or probabilities; we also demonstrated how such quantities could be added to the data in the tuple space or to the processes for scheduling parallel threads. We have also shown how to add mobility, as in the KLAIM language. We introduced probabilities both at the local (or process) level and at the network level. We also presented a continuous-time model where we use rates to determine how often a node is active. This information contributes to the probability of network updates.

The probabilistic version of KLAIM we have introduced in this paper is closely related to various *probabilistic programming languages* and *probabilistic process calculi* proposed in the recent literature. Among these we mention discrete time approaches — e.g. PCCS [29,30], PCCP [31], etc. — as well as continuous time approaches — e.g. PEPA [11], Stochastic π calculus [32].

Work in performance analysis is often based on probabilistic process calculi, for example, on Hillston's PEPA [33], or EMPA by Bernardo and Gorrieri [34]. One of the long term aims of the work presented in this paper is the development of semantics based approaches towards *performance analysis* along similar lines as in classical program analysis. We also aim to investigate more closely the relation of our work to recent work on probabilistic verification and model checking, such as PRISM [35] and de Alfaro [36].

We have considered here a model based on Poisson processes which are some of the simplest examples of continuous-time Markov chains. More complicated continuous time behaviour could be considered, but this might require more parameters than just rate to describe the time distributions [14]. The language could also be extended so as to allow for a dynamic change of probabilities and rates, i.e. for rate and probability which depend on the time. These last two extensions require further work.

References

1. Gelernter, D.: Generative communication in linda. ACM Trans. Program. Lang. Syst. **7** (1985) 80–112
2. Gelernter, D., Carriero, N.: Coordination languages and their significance. Commun. ACM **35** (1992) 97–107
3. Arbab, F.: Manifold. Future Generation Computer Systems **10** (1994) 273–277
4. Sands, D., Weichert, M.: From gamma to cbs: Refining multiset transformations with broadcasting processes. In El-Rewini, H., ed.: Proceedings of 31st Hawaii International Conference on System Sciences. Volume VII., IEEE (1998) 265–274
5. Bravetti, M., Gorrieri, R., Lucchi, R., Zavattaro, G.: Quantitative information in the tuple space coordination model. Theoretical Computer Science (To appear)
6. Di Pierro, A., Hankin, C., Wiklicky, H.: Probabilistic KLAIM. In Nicola, R.D., Ferrari, G., Meredith, G., eds.: Proceedings of Coordination 2004. Number 2949 in Lecture Notes in Computer Science, Berlin — Heidelberg — New York, Springer Verlag (2004) 119–134

7. Di Pierro, A., Hankin, C., Wiklicky, H.: Continuous-time probabilistic KLAIM. In: SecCo'04 — CONCUR Workshop on Security Issues in Coordination Models, Languages, and Systems. Electronic Notes in Theoretical Computer Science, Elsevier (2004)
8. Nicola, R.D., Latella, D., Massink, M.: Formal modeling and quantitative analysis of KLAIM-based mobile systems. In: 20th Annual ACM Symposium on Applied Computing, ACM (2005)
9. van Glabbeek, R., Smolka, S., Steffen, B.: Reactive, generative and stratified models of probabilistic processes. Information and Computation **121** (1995) 59–80
10. De Nicola, R., Ferrari, G., Pugliese, R.: KLAIM: A kernel language for agents interaction and mobility. IEEE Transactions on Software Engineering **24** (1998) 315–330
11. Hillston, J.: PEPA: Performance enhanced process algebra. Technical Report CSR-24-93, University of Edinburgh, Edinburgh, Scotland (1993)
12. Tijms, H.C.: Stochastic Models – An Algorithmic Approach. John Wiley & Sons, Chichester (1994)
13. Norris, J.: Markov Chains. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, Cambridge (1997)
14. Bause, F., Kritzinger, P.S.: Stochastic Petri Nets – An Introduction to the Theory. second edn. Vieweg Verlag (2002)
15. Cousot, P., Cousot, R.: Abstract Interpretation and Applications to Logic Programs. Journal of Logic Programming **13** (1992) 103–180
16. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Verlag, Berlin – Heidelberg (1999)
17. Di Pierro, A., Wiklicky, H.: Concurrent Constraint Programming: Towards Probabilistic Abstract Interpretation. In: Proceedings of PPDP'00, Montréal, Canada, ACM (2000) 127–138
18. Di Pierro, A., Wiklicky, H.: Linear structures for concurrency in probabilistic programming languages. In: Proceedings of MFCSIT00 – First Irish Conference on the Mathematical Foundations of Computer Science and Information Technology, Cork, Ireland. Volume 40 of Electronic Notes in Theoretical Computer Science., Elsevier (2001)
19. Di Pierro, A., Wiklicky, H.: Operator algebras and the operational semantics of probabilistic languages. In: Proceedings of MFCSIT04 – Third Irish Conference on the Mathematical Foundations of Computer Science and Information Technology, Dublin, Ireland. Electronic Notes in Theoretical Computer Science, Elsevier (To appear)
20. Böttcher, A., Silbermann, B.: Introduction to Large Truncated Toeplitz Matrices. Springer Verlag, New York (1999)
21. Deutsch, F.: Best Approximation in Inner Product Spaces. Volume 7 of CMS Books in Mathematics. Springer Verlag, New York — Berlin (2001)
22. Di Pierro, A., Hankin, C., Wiklicky, H.: Measuring the confinement of probabilistic systems. Theoretical Computer Science **340** (2005) 3–56
23. Ben-Israel, A., Greville, T.: Generalised Inverses — Theory and Applications. second edn. Volume 15 of CMS Books in Mathematics. Springer Verlag, New York — Berlin (2003)
24. Di Pierro, A., Hankin, C., Wiklicky, H.: Quantitative relations and approximate process equivalences. In Lugiez, D., ed.: Proceedings of CONCUR'03 — International Conference on Concurrency Theory. Number 2761 in Lecture Notes in Computer Science, Berlin — Heidelberg — New York, Springer Verlag (2003) 508–522

25. Di Pierro, A., Hankin, C., Wiklicky, H.: Quantitative static analysis of distributed systems. Journal of Functional Programming **15** (2005) 1–47
26. Friedberg, S., Insel, A., Spence, L.: Linear Algebra. forth edn. Prentice Hall (2003)
27. Prasolov, V.: Problems and Theorems in Linear Algebra. Volume 134 of Translation of Mathematical Monographs. American Mathematical Society, Providence, Rhode Island (1994)
28. Hirsch, M., Smale, S.: Differential Equations, Dynamical Systems, and Linear Algebra. Academic Press, Orlando (1974)
29. Giacalone, A., Jou, C.C., Smolka, S.: Algebraic reasoning for probabilistic concurrent systems. In: Proceedings of the IFIP WG 2.2/2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, North-Holland (1990) 443–458
30. Jonsson, B., Yi, W., Larsen, K.: 11. In: Probabilistic Extensions of Process Algebras. Elsevier Science, Amsterdam (2001) 685–710 see [**?**].
31. Di Pierro, A., Wiklicky, H.: Quantitative observables and averages in Probabilistic Concurrent Constraint Programming. In Apt, K., Kakas, T., Monfroy, E., Rossi, F., eds.: New Trends in Constraints — Selected Papers of the ERCIM/Compulog Workshop on Constraints, October 1999, Paphos, Cyprus. Number 1865 in Lecture Notes in Computer Science, Berlin — Heidelberg — New York, Springer Verlag (2000)
32. Priami, C.: Stochastic π-calculus. Computer Journal **38** (1995) 578–589
33. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press (1996)
34. Bernardo, M., Gorrieri, R.: A tutorial on empa: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. Technical Report UBLCS-96-17, Department of Computer Science, University of Bologna (1997)
35. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: A hybrid approach. In Katoen, J.P., Stevens, P., eds.: Proceedings of TACAS'02. Volume 2280 of Lecture Notes in Computer Science., Springer Verlag (2002) 52–66
36. de Alfaro, L.: Formal Verification of Probabilistic Systems. PhD thesis, Stanford University, Department of Computer Science (1998)

Games with Secure Equilibria*,**

Krishnendu Chatterjee, Thomas A. Henzinger, and Marcin Jurdziński

Department of Electrical Engineering and Computer Sciences,
University of California, Berkeley, USA
{c_krish, tah, mju}@eecs.berkeley.edu

Abstract. In 2-player non-zero-sum games, Nash equilibria capture the options for rational behavior if each player attempts to maximize her payoff. In contrast to classical game theory, we consider lexicographic objectives: first, each player tries to maximize her own payoff, and then, the player tries to minimize the opponent's payoff. Such objectives arise naturally in the verification of systems with multiple components. There, instead of proving that each component satisfies its specification no matter how the other components behave, it often suffices to prove that each component satisfies its specification provided that the other components satisfy their specifications. We say that a Nash equilibrium is *secure* if it is an equilibrium with respect to the lexicographic objectives of both players. We prove that in graph games with Borel winning conditions, which include the games that arise in verification, there may be several Nash equilibria, but there is always a unique maximal payoff profile of a secure equilibrium. We show how this equilibrium can be computed in the case of ω-regular winning conditions, and we characterize the memory requirements of strategies that achieve the equilibrium.

1 Introduction

We consider 2-player non-zero-sum games, i.e., non-strictly competitive games. A possible behavior of the two players is captured by a strategy profile (σ, π), where σ is a strategy of player 1, and π is a strategy of player 2. Classically, the behavior (σ, π) is considered *rational* if the strategy profile is a Nash equilibrium [7] — that is, if neither player can increase her payoff by unilaterally changing her strategy. Formally, let $v_1^{\sigma,\pi}$ be the real-valued payoff of player 1 if the strategies (σ, π) are played, and let $v_2^{\sigma,\pi}$ be the corresponding payoff of player 2. Then (σ, π) is a Nash equilibrium if (1) $v_1^{\sigma,\pi} \geq v_1^{\sigma',\pi}$ for all player 1 strategies σ', and (2) $v_2^{\sigma,\pi} \geq v_2^{\sigma,\pi'}$ for all player 2 strategies π'. Nash equilibria formalize a notion of rationality which is strictly *internal*: each player cares about her own payoff but does not in the least care (cooperatively or adversarially) about the other player's payoff.

* This research was supported in part by the ONR grant N00014-02-1-0671, the AFOSR MURI grant F49620-00-1-0327, and the NSF grant CCR-0225610.

** This is an extended version of the paper "Games with Secure Equilibria" that appeared in the proceedings of *Logic in Computer Science* (LICS), 2004.

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 141–161, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Choosing Among Nash Equilibria. A classical problem is that many games have *multiple* Nash equilibria, and some of them may be preferable to others. For example, one might partially order the equilibria by $(\sigma, \pi) \succeq (\sigma', \pi')$ if both $v_1^{\sigma,\pi} \geq v_1^{\sigma',\pi'}$ and $v_2^{\sigma,\pi} \geq v_2^{\sigma',\pi'}$. If a *unique maximal* Nash equilibrium exists in this order, then it is preferable for both players. However, maximal Nash equilibria may not be unique. In these cases *external* criteria, such as the sum of the payoffs for both players, have been used to evaluate different rational behaviors [9,14]. These external criteria, which are based on a single preference order on strategy profiles, are *cooperative*, in that they capture social aspects of rational behavior. We define and study, for the first time, an *adversarial* external criterion for rational behavior. Put simply, we assume that each player attempts to minimize the other player's payoff as long as, by doing so, she does not decrease her own payoff. This yields two different preference orders on strategy profiles, one for each player, and gives rise to a new notion of equilibrium.

Adversarial External Choice. According to our notion of rationality, among two strategy profiles (σ, π) and (σ', π'), player 1 prefers (σ, π), denoted $(\sigma, \pi) \succeq_1 (\sigma', \pi')$, if either $v_1^{\sigma,\pi} > v_1^{\sigma',\pi'}$, or both $v_1^{\sigma,\pi} = v_1^{\sigma',\pi'}$ and $v_2^{\sigma,\pi} \leq v_2^{\sigma',\pi'}$. In other words, the *preference order* $\succeq_1$ of player 1 is lexicographic: the primary goal of player 1 is to maximize her own payoff; the secondary goal is to minimize the opponent's payoff. The preference order $\succeq_2$ of player 2 is defined symmetrically. It should be noted that, defined in this way, adversarial external choice cannot be internalized uniformly over all games by changing the payoff functions of the two players: if $v_1^{\sigma,\pi} = v_1^{\sigma',\pi'}$ and $v_2^{\sigma,\pi} \leq v_2^{\sigma',\pi'}$, then uniform internalization would require to increase $v_1^{\sigma,\pi}$ by an arbitrarily small $\varepsilon > 0$.

Secure Equilibria. The two orders $\succeq_1$ and $\succeq_2$ on strategy profiles, which express the preferences of the two players, induce the following refinement of the Nash equilibrium notion: (σ, π) is a *secure equilibrium* if (1) $(v_1^{\sigma,\pi}, v_2^{\sigma,\pi}) \succeq_1 (v_1^{\sigma',\pi}, v_2^{\sigma',\pi})$ for all player 1 strategies σ', and (2) $(v_1^{\sigma,\pi}, v_2^{\sigma,\pi}) \succeq_2 (v_1^{\sigma,\pi'}, v_2^{\sigma,\pi'})$ for all player 2 strategies π'. Note that every secure equilibrium is a Nash equilibrium, but a Nash equilibrium need not be secure. The name "secure" equilibrium derives from the following equivalent characterization. We say that a strategy profile (σ, π) is *secure* if any rational deviation of player 2 —i.e., a deviation that does not decrease her payoff— will not decrease the payoff of player 1, and symmetrically, any rational deviation of player 1 will not decrease the payoff of player 2. Formally, (σ, π) is secure if for all player 2 strategies π', if $v_2^{\sigma,\pi'} \geq v_2^{\sigma,\pi}$ then $v_1^{\sigma,\pi'} \geq v_1^{\sigma,\pi}$, and for all player 1 strategies σ', if $v_1^{\sigma',\pi} \geq v_1^{\sigma,\pi}$ then $v_2^{\sigma',\pi} \geq v_2^{\sigma,\pi}$. The secure profile (σ, π) can thus be interpreted as a contract between the two players which enforces cooperation: any unilateral selfish deviation by one player cannot put the other player at a disadvantage if she follows the contract. It is not difficult to show (see Section 2) that a strategy profile is a secure equilibrium iff it is both a secure profile and a Nash equilibrium. Thus, the secure equilibria are those Nash equilibria which represent enforceable contracts between the two players.

Motivation: Verification of Component-Based Systems. The motivation for our definitions comes from verification. There, one would like to prove that a component of a system (player 1) can satisfy a specification no matter how the environment (player 2) behaves [3]. Classically, this is modeled as a strictly competitive (zero-sum) game, where the environment's objective is the complement of the component's objective. However, the zero-sum model is often naive, as the environment itself typically consists of components, each with its own specification (i.e., objective). Moreover, the individual component specifications are usually not complementary; a common example is that each component must maintain a local invariant. So a more appropriate approach is to prove that player 1 can meet her objective no matter how player 2 behaves *as long as* player 2 does not sabotage her own objective. In other words, classical correctness proofs of a component assume absolute worst-case behavior of the environment, while it would suffice to assume only *relative* worst-case behavior of the environment —namely, relative to the assumption that the environment itself is correct (i.e., meets its specification). Such relative worst-case reasoning called assume-guarantee reasoning [1,2,13] so far has not been studied in the natural setting offered by game theory.

Existence and Uniqueness of Maximal Secure Equilibria. We will see that in general games, such as matrix games, there may be multiple secure equilibrium payoff profiles, even several incomparable maximal ones. However, the games that occur in verification have a special form. They are played on directed graphs whose nodes represent system states, and whose edges represent system transitions. The nodes partitioned into two sets: in player 1 nodes, the first player chooses an outgoing edge, and in player 2 nodes, the second player chooses an outgoing edge. By repeating these choices ad infinitum, an infinite path through the graph is formed, which represents a system trace. The objective φ_i of each player i is a set of infinite paths; for example, an invariant (or "safety") objective is the set of infinite paths that do not visit unsafe states. Each player i attempts to satisfy her objective φ_i by choosing a strategy that ensures that the outcome of the game lies in the set φ_i. The objective φ_i is typically an ω-regular set (specified, e.g., in temporal logic), or more generally, a Borel set [8] in the Cantor topology on infinite paths. We call these games 2-player non-zero-sum *graph games with Borel objectives*. Our main result shows that for these games, which may have multiple maximal Nash equilibria, there always exists a unique maximal secure equilibrium payoff profile. In other words, in graph games with Borel objectives there is a compelling notion of rational behavior for each player, which is (1) a classical Nash equilibrium, (2) an enforceable contract ("secure"), and (3) a guarantee of maximal payoff for each player among all behaviors that achieve (1) and (2).

Examples. Consider the game graph shown in Fig. 1. Player 1 chooses the successor node at square nodes and her objective is to reach the target s_4, a reachability (co-safety) objective. Player 2 chooses the successor node at diamond nodes and her objective is to reach s_3 or s_4, also a reachability objective. There are two player 1 strategies: σ_1 chooses the move $s_0 \to s_1$, and σ_2 chooses $s_0 \to s_2$.

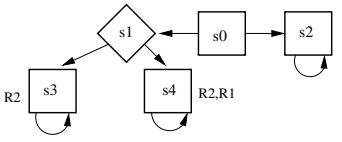


Fig. 1. A graph game with reachability objectives

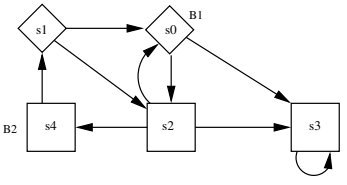


Fig. 2. A graph game with Büchi objectives

There are also two player 2 strategies: π_1 chooses $s_1 \to s_3$, and π_2 chooses $s_1 \to s_4$. The strategy profile (σ_1, π_1) leads the game into s_3 and therefore gives the payoff profile (0,1), meaning player 1 loses and player 2 wins (i.e., only player 2 reaches her target). The strategy profiles (σ_1, π_2), (σ_2, π_1), and (σ_2, π_2) give the payoffs (1,1), (0,0), and (0,0), respectively. All four strategy profiles are Nash equilibria; for example, in (σ_1, π_1) player 1 does not have an incentive to switch to strategy σ_2 (which would still give her payoff 0), and neither does player 2 have an incentive to switch to π_2 (she is already getting payoff 1). However, the strategy profile (σ_1, π_1) is not a secure equilibrium, because player 2 can lower player 1's payoff (from 1 to 0) without changing her own payoff by switching to strategy σ_2. Similarly, the strategy profile (σ_1, π_2) is not secure, because player 1 can lower player 2's payoff without changing her own payoff by switching to σ_1. So if both players, in addition to maximizing their own payoff, also attempt to minimize the opponents payoff, then the resulting payoff profile is unique, namely, (0,0). In other words, in this game, the only rational behavior for both players is to deny each other's objectives.

This is not always the case: sometimes it is beneficial for both players to cooperate to achieve their own objectives, with the result that both players win. Consider the game graph shown in Fig. 2. Both players have Büchi objectives: player 1 (square) wants to visit s_0 infinitely often, and player 2 (diamond) wants to visit s_4 infinitely often. If player 2 always chooses $s_1 \to s_0$ and player 1 always chooses $s_2 \to s_4$, then both players win. This Nash equilibrium is also secure: if player 1 deviates by choosing $s_2 \to s_0$, then player 2 can "retaliate" by choosing $s_0 \to s_3$; similarly, if player 2 deviates by choosing $s_1 \to s_2$, then player 2 can retaliate by $s_2 \to s_3$. It follows that for purely selfish motives (and not some social reason), both players have an incentive to cooperate to achieve the maximal secure equilibrium payoff (1,1).

Outline and Results. In Section 2, we define the notion of secure equilibrium and give several interpretations through alternative definitions. In Section 3 we

prove the existence and uniqueness of maximal secure equilibria in graph games with Borel objectives. The proof is based on the following classification of strategies. A player 1 strategy is called *strongly winning* if it ensures that player 1 wins and player 2 loses (i.e., the outcome of the game satisfies $\varphi_1 \wedge \neg\varphi_2$). A player 1 strategy is *retaliating* if it ensures that player 1 wins if player 2 wins (i.e., the outcome satisfies $\varphi_2 \rightarrow \varphi_1$). In other words, a retaliating strategy for player 1 ensures that if player 2 causes player 1 to lose, then player 2 will lose too. If both players follow retaliating strategies (σ, π), they may both win —in this case, we say that (σ, π) is a *winning pair of retaliating strategies*— or they may both lose. We show that at every node of a graph game with Borel objectives, either one of the two players has a strongly winning strategy, or there is a pair of retaliating strategies. Based on this insight, we give an algorithm for computing the secure equilibria in graph games in the case that both players' objectives are ω-regular. In Section 4, we analyze the memory requirements of strongly winning and retaliating strategies in graph games with ω-regular objectives. Our results (in Table 1 and 2) consider safety, reachability, Büchi, co-Büchi, and general parity objectives. We show that strongly winning and retaliating strategies often require memory, even in the simple case that a player pursues a reachability objective. In Section 5, we generalize the notion of secure equilibria from 2-player to n-player games. We show that there can be multiple maximal secure equilibria in 3-player graph games with reachability objectives.

2 Definitions

In a secure game the objective of player 1 is to maximize her own payoff and then minimize the payoff of player 2. Similarly, player 2 maximizes her own payoff and then minimizes the payoff of player 1. We want to determine the best payoff that each player can ensure when both players play according to these preferences. We formalize this as follows. A *strategy profile* (σ, π) is a pair of strategies, where σ is a player 1 strategy and π is a player 2 strategy. The strategy profile (σ, π) gives rise to a *payoff profile* $(v_1^{\sigma,\pi}, v_2^{\sigma,\pi})$, where $v_1^{\sigma,\pi}$ is the payoff of player 1 if the two players follow the strategies σ and π respectively, and $v_2^{\sigma,\pi}$ is the corresponding payoff of player 2. We define the player 1 preference order $\preceq_1$ and the player 2 preference order $\preceq_2$ on payoff profiles lexicographically:

$$(v_1, v_2) \prec_1 (v_1', v_2') \text{ iff } (v_1 < v_1') \vee (v_1 = v_1' \wedge v_2 > v_2'),$$

that is, player 1 prefers a payoff profile which gives her greater payoff, and if two payoff profiles match in the first component, then she prefers the payoff profile in which the player 2's payoff is minimized; symmetrically,

$$(v_1, v_2) \prec_2 (v_1', v_2') \text{ iff } (v_2 < v_2') \vee (v_2 = v_2' \wedge v_1 > v_1').$$

Given two payoff profiles (v_1, v_2) and (v_1', v_2'), we write $(v_1, v_2) = (v_1', v_2')$ iff $v_1 = v_1'$ and $v_2 = v_2'$, and $(v_1, v_2) \preceq_1 (v_1, v_2')$ iff either $(v_1, v_2) \prec_1 (v_1', v_2')$ or $(v_1, v_2) = (v_1', v_2')$. We define $\preceq_2$ analogously.

Definition 1 (Secure strategy profiles). *A strategy profile* (σ, π) *is* secure *if the following two conditions hold:*

$$\forall \pi'.\ (v_1^{\sigma,\pi'} < v_1^{\sigma,\pi}) \rightarrow (v_2^{\sigma,\pi'} < v_2^{\sigma,\pi})$$

$$\forall \sigma'.\ (v_2^{\sigma',\pi} < v_2^{\sigma,\pi}) \rightarrow (v_1^{\sigma',\pi} < v_1^{\sigma,\pi})$$ ∎

A secure strategy for player 1 ensures that if player 2 tries to decrease player 1's payoff, then player 2's payoff decreases as well, and vice versa.

Definition 2 (Secure equilibria). *A strategy profile* (σ, π) *is a* secure equilibrium *if the strategy profile is a Nash equilibrium and it is secure.* ∎

Lemma 1 (Equivalent characterization). *The strategy profile* (σ, π) *is a secure equilibrium iff the following two conditions hold:*

$$\forall \pi'.\ (v_1^{\sigma,\pi'}, v_2^{\sigma,\pi'}) \preceq_2 (v_1^{\sigma,\pi}, v_2^{\sigma,\pi})$$

$$\forall \sigma'.\ (v_1^{\sigma',\pi}, v_2^{\sigma',\pi}) \preceq_1 (v_1^{\sigma,\pi}, v_2^{\sigma,\pi})$$

Proof. Given (σ, π) is a Nash equilibrium strategy profile we have for all π', $v_2^{\sigma,\pi'} \leq v_2^{\sigma,\pi}$. Since the strategy profile is also a secure strategy profile for all strategy π' we have $(v_1^{\sigma,\pi'} < v_1^{\sigma,\pi}) \rightarrow (v_2^{\sigma,\pi'} < v_2^{\sigma,\pi})$. It follows from above that for any arbitrary π' the following condition hold:

$$(v_2^{\sigma,\pi'} = v_2^{\sigma,\pi}\ \wedge\ v_1^{\sigma,\pi} \leq v_1^{\sigma,\pi'}) \vee (v_2^{\sigma,\pi'} < v_2^{\sigma,\pi}).$$

Hence for all π' we have $(v_1^{\sigma,\pi'}, v_2^{\sigma,\pi'}) \preceq_2 (v_1^{\sigma,\pi}, v_2^{\sigma,\pi})$. The argument for the other case is symmetric.

Hence neither player 1 nor player 2 has any incentive to switch from the strategy profile (σ, π) to increase the payoff profile according to their respective payoff profile ordering. ∎

Example 1 (Matrix games). A secure equilibrium need not exist in a matrix game. We give an example of a matrix game where no Nash equilibrium is secure. Consider the game M_1 below, where the row player can choose row 1 or row 2 (denoted r_1 and r_2, respectively), and the column player chooses between the two columns (denoted c_1 and c_2). The first component of the payoff is the row player payoff, and the second component is the column player payoff.

$$M_1 = \begin{bmatrix} (3,3) & (1,3) \\ (3,1) & (2,2) \end{bmatrix}$$

In this game the strategy profile (r_1, c_1) is the only Nash equilibrium. But (r_1, c_1) is not a secure strategy profile, because if the row player plays r_1, then the column player playing c_2 can still get payoff 3 and decrease the row player's payoff to 1. In the game M_2 there are two Nash equilibria, namely, (r_1, c_2) and (r_2, c_1), and the strategy profile (r_2, c_1) is a secure strategy profile as well. Hence the strategy profile (r_2, c_1) is a secure equilibrium. However the strategy profile (r_1, c_2) is not secure.

$$M_2 = \begin{bmatrix} (0,0) & (1,0) \\ (\frac{1}{2},\frac{1}{2}) & (\frac{1}{2},\frac{1}{2}) \end{bmatrix}$$

Multiple secure equilibria can exist, as in the case, for example, in a matrix game where all entries of the matrix are the same. We now present an example of a matrix game with multiple secure equilibria profile. Consider the following matrix game M_3. The strategy profile (r_1, c_1) and (r_2, c_2) are both secure equilibria. The former has a payoff profile $(2, 1)$ and the later has a payoff profile $(1, 2)$. Hence there can be multiple secure equilibria payoff profiles and in case there are multiple secure equilibria payoff profiles the maximal payoff profile is not always unique.

$$M_3 = \begin{bmatrix} (2,1) & (0,0) \\ (0,0) & (1,2) \end{bmatrix}$$ ∎

3 2-Player Non-zero-sum Games on Graphs

We consider 2-player infinite path-forming games played on graphs. A *game graph* $G = ((V, E), (V_1, V_2))$ consists of a directed graph (V, E), where V is the set of states (vertices) and E is the set of edges, and a partition (V_1, V_2) of the states. For technical convenience we assume that every state has at least one outgoing edge. The two players, player 1 and player 2, keep moving a token along the edges of the game graph: player 1 moves the token from states in V_1, and player 2 moves the token from states in V_2. A *play* is an infinite path $\Omega = \langle s_0, s_1, s_2, \ldots \rangle$ through the game graph, that is, $(s_k, s_{k+1}) \in E$ for all $k \geq 0$. A strategy for player 1, given a prefix of a play (i.e., a finite sequence of states), specifies a next state to extend the play. Formally, a *strategy* for player 1 is a function $\sigma\colon V^* \cdot V_1 \to V$ such that for all $x \in V^*$ and $s \in V_1$, we have $(s, \sigma(x \cdot s)) \in E$. A strategy π for player 2 is defined symmetrically. We write Σ and Π to denote the sets of strategies for player 1 and player 2, respectively. A strategy is memoryless if it is independent of the history of play. Formally, a strategy τ of player i, where $i \in \{1, 2\}$, is *memoryless* if $\tau(x \cdot s) = \tau(x' \cdot s)$ for all $x, x' \in V^*$ and all $s \in V_i$; hence a memoryless strategy of player i can be represented as a function $\tau\colon V_i \to V$. A play $\Omega = \langle s_0, s_1, s_2, \ldots \rangle$ is *consistent* with a strategy τ of player i if for all $k \geq 0$, if $s_k \in V_i$, then $s_{k+1} = \tau(s_0, s_1, \ldots, s_k)$. Given a state $s \in V$, a strategy σ of player 1, and a strategy π of player 2, there is a unique play $\Omega_{\sigma,\pi}(s)$, the *outcome* of the game, which starts from s and is consistent with both σ and π.

Objectives of the players are specified generally as sets $\varphi \subseteq V^\omega$ of infinite paths. We write $\Omega \models \varphi$ instead of $\Omega \in \varphi$ for infinite paths Ω and objectives φ. We use boolean operators such as $\vee$, $\wedge$, and $\neg$ on objectives to denote set union, intersection, and complement. A *Borel objective* is a Borel set $\varphi \subseteq V^\omega$ in the Cantor topology on V^ω. The following celebrated result of Martin establishes that all games with Borel objectives are determined.

Theorem 1 (Borel determinacy [11]). *For every 2-player graph game G, every state s, and every Borel objective φ, either (1) there is a strategy σ of*

player 1 such that for all strategies π' of player 2, we have $\Omega_{\sigma,\pi'}(s) \models \varphi$, or (2) there is a strategy π of player 2 such that for all strategies σ' of player 1, we have $\Omega_{\sigma',\pi}(s) \models \neg\varphi$.

In verification, objectives are usually *ω-regular* sets. The ω-regular sets occur in the low levels of the Borel hierarchy (in $\Sigma_3 \cap \Pi_3$) and form a robust and expressive class for determining the payoffs of commonly used system specifications [10,16].

We consider non-zero-sum games on graphs. For our purposes, a *graph game* $(G, s, \varphi_1, \varphi_2)$ consists of a game graph G, say with state set V, together with a start state $s \in V$ and two Borel objectives $\varphi_1, \varphi_2 \subseteq V^\omega$. The game starts at state s, player 1 pursues the objective φ_1, and player 2 pursues the objective φ_2 (in general, φ_2 is not the complement of φ_1). Player $i \in \{1, 2\}$ gets payoff 1 if the outcome of the game is a member of φ_i, and she gets payoff 0 otherwise. In the following, we fix the game graph G and the objectives φ_1 and φ_2, but we vary the start state s of the game. Thus we parameterize the payoffs by s: given strategies σ and π for the two players, we write $v_i^{\sigma,\pi}(s) = 1$ if $\Omega_{\sigma,\pi}(s) \models \varphi_i$, and $v_i^{\sigma,\pi}(s) = 0$ otherwise, for $i \in \{1, 2\}$. Similarly, we sometimes refer to Nash equilibria and secure strategy profiles of the graph game $(G, s, \varphi_1, \varphi_2)$ as equilibria and secure profiles *at* the state s.

3.1 Unique Maximal Secure Equilibria

Consider a game graph G with state set V, and Borel objectives φ_1 and φ_2 for the two players.

Definition 3 (Maximal secure equilibria). *For $v, w \in \{0, 1\}$, we write $S_{v,w} \subseteq V$ to denote the set of states s such that a secure equilibrium with the payoff profile (v, w) exists in the game $(G, s, \varphi_1, \varphi_2)$, that is, $s \in S_{v,w}$ iff there is a secure equilibrium (σ, π) at s such that $(v_1^{\sigma,\pi}(s), v_2^{\sigma,\pi}(s)) = (v, w)$. Similarly, $MS_{v,w} \subseteq S_{v,w}$ denotes the set of states s such that the payoff profile (v, w) is a* maximal secure equilibrium payoff profile *at s, that is, $s \in MS_{v,w}$ iff (1) $s \in S_{v,w}$ and (2) for all $v', w' \in \{0, 1\}$, if $s \in S_{v',w'}$, then $(v', w') \preceq_1 (v, w)$ and $(v', w') \preceq_2 (v, w)$.* ∎

We now define the notions of strongly winning and retaliating strategies, which capture the essence of secure equilibria. A strategy for player 1 is strongly winning if it ensures that the objective of player 1 is satisfied and the objective of player 2 is not. A retaliating strategy for player 1 ensures that for every strategy of player 2, if the objective of player 2 is satisfied, then the objective of player 1 is satisfied as well. We will show that every secure equilibrium either contains a strongly winning strategy for one of the players, or it consists of a pair of retaliating strategies.

Definition 4 (Strongly winning strategies). *A strategy σ is* strongly winning *for player 1 from a state s if she can ensure the payoff profile $(1, 0)$ in the game $(G, s, \varphi_1, \varphi_2)$ by playing the strategy σ. Formally, σ is strongly winning*

for player 1 if for all player 2 strategies π, we have $\Omega_{\sigma,\pi}(s) \models (\varphi_1 \wedge \neg\varphi_2)$. The strongly winning strategies for player 2 are defined symmetrically. ∎

Definition 5 (Retaliating strategies). *A strategy σ is a* retaliating *strategy for player 1 from a state s if for all player 2 strategies π, we have $\Omega_{\sigma,\pi}(s) \models (\varphi_2 \rightarrow \varphi_1)$. Similarly, a strategy π is a retaliating strategy for player 2 from s if for all player 1 strategies σ, we have $\Omega_{\sigma,\pi}(s) \models (\varphi_1 \rightarrow \varphi_2)$. We write $Re_1(s)$ and $Re_2(s)$ to denote the sets of retaliating strategies for player 1 and player 2 from s. A strategy profile (σ, π) is a* retaliation strategy profile *at a state s if both σ and π are retaliating strategies from s.* ∎

Example 2 (Büchi-Büchi game). Recall the game shown in Fig. 2. Consider the memoryless strategies of player 2 at state s_0. If player 2 chooses $s_0 \rightarrow s_3$, then player 2 does not satisfy her Büchi objective. If player 2 chooses $s_0 \rightarrow s_2$, then at state s_2 player 1 chooses $s_2 \rightarrow s_0$, and hence player 1's objective is satisfied, but player 2's objective is not satisfied. Thus, no memoryless strategy for player 2 can be a winning retaliating strategy at s_0.

Now consider the strategy π_g for player 2 which chooses $s_0 \rightarrow s_2$ if between the last two consecutive visits to s_0 the state s_4 was visited, and otherwise it chooses $s_0 \rightarrow s_3$. Given this strategy, for every strategy of player 1 that satisfies player 1's objective, player 2's objective is also satisfied. Let σ_g be the player 1 strategy that chooses $s_2 \rightarrow s_4$ if between the last two consecutive visits to s_2 the state s_0 was visited, and otherwise chooses $s_2 \rightarrow s_3$. The strategy profile (σ_g, π_g) consists of a pair of winning retaliating strategies, as it satisfies the Büchi objectives of both players. If instead, player 2 always chooses $s_0 \rightarrow s_3$, and player 1 always chooses $s_2 \rightarrow s_3$, we obtain a memoryless retaliation strategy profile, which is not winning for either player: it is a Nash equilibrium at state s_0 with the payoff profile $(0, 0)$. Finally, suppose that at s_0 player 2 always chooses s_2, and at s_2 player 1 always chooses s_0. This strategy profile is again a Nash equilibrium, with the payoff profile $(0, 1)$ at s_0, but not a retaliation strategy profile. This shows that at state s_0 the Nash equilibrium payoff profiles $(0, 1)$, $(0, 0)$, and $(1, 1)$ are possible, but only $(0, 0)$ and $(1, 1)$ are secure. ∎

Definition 6 (Winning sets). *We define the following state sets in terms of strongly winning and retaliating strategies.*

- *The sets of states where player 1 or player 2 have a strongly winning strategy, denoted by W_{10} and W_{01}, respectively:*

$$W_{10} = \{\, s \in V \,:\, \exists \sigma \in \Sigma.\ \forall \pi \in \Pi.\ \ \Omega_{\sigma,\pi}(s) \models (\varphi_1 \wedge \neg\varphi_2) \,\}$$

$$W_{01} = \{\, s \in V \,:\, \exists \pi \in \Pi.\ \forall \sigma \in \Sigma.\ \ \Omega_{\sigma,\pi}(s) \models (\varphi_2 \wedge \neg\varphi_1) \,\}$$

- *The set of states where both players have retaliating strategies and there exists a retaliation strategy profile whose strategies satisfy the objectives of both players:*

$$W_{11} = \{\, s \in V \,:\, \exists \sigma \in Re_1(s).\ \exists \pi \in Re_2(s).\ \ \Omega_{\sigma,\pi}(s) \models (\varphi_1 \wedge \varphi_2) \,\}$$

– *The set of states where both players have retaliating strategies and for every retaliation strategy profile, neither the objective of player 1 nor the objective of player 2 is satisfied:*

$$W_{00} = \{\, s \in V \,:\, Re_1(s) \neq \emptyset \;\; \textit{and} \;\; Re_2(s) \neq \emptyset \;\; \textit{and} \\ \forall \sigma \in Re_1(s).\, \forall \pi \in Re_2(s).\;\; \Omega_{\sigma,\pi}(s) \models (\neg\varphi_1 \wedge \neg\varphi_2) \,\} \qquad \blacksquare$$

We show that the four sets W_{10}, W_{01}, W_{11}, and W_{00} form a partition of the state space. This result fully characterizes each state of a 2-player non-zero-sum graph game with Borel objectives, just like the determinacy result (Theorem 1) fully characterizes the zero-sum case. In the zero-sum case, where $\varphi_2 = \neg\varphi_1$, the sets W_{10} and W_{01} specify the winning states for players 1 and 2, respectively, $W_{11} = \emptyset$ by definition, and $W_{00} = \emptyset$ by determinacy. We also show that for all $v, w \in \{0, 1\}$, we have $MS_{v,w} = W_{v,w}$. It follows that for 2-player graph games (1) secure equilibria always exist, and moreover, (2) there is always a unique maximal secure equilibrium payoff profile. (Example 2 showed that there can be multiple secure equilibria with different payoff profiles). The proof proceeds in several steps.

Lemma 2. $W_{10} = \{\, s \in V \,:\, Re_2(s) = \emptyset \,\}$ *and* $W_{01} = \{\, s \in V \,:\, Re_1(s) = \emptyset \,\}$.

Proof. We show the inclusion of one set in the other for both the direction:

1. $W_{10} \subseteq \{\, s : Re_2(s) = \emptyset \,\}$ as a strongly winning strategy σ of player 1 to satisfy $(\varphi_1 \wedge \neg\varphi_2)$ against any strategy π of player 2 is a witness to exhibit that there is no retaliation strategy for player 2.
2. It follows from Borel determinacy (Theorem 1) that from every state s in $V \setminus W_{10}$ there is a strategy π for player 2 to satisfy $(\neg\varphi_1 \vee \varphi_2)$ against any strategy of player 1. The strategy π is a retaliation strategy for player 2. Hence we have $V \setminus W_{10} \subseteq \{\, s : Re_2(s) \neq \emptyset \,\}$.

The claim is a consequence of the above facts. $\blacksquare$

Lemma 3. *Consider the following sets:*

$$T_1 = \{\, s \in V \,:\, \forall \sigma \in Re_1(s).\, \forall \pi \in Re_2(s).\;\; \Omega_{\sigma,\pi}(s) \models (\neg\varphi_1 \wedge \neg\varphi_2) \,\}$$

$$T_2 = \{\, s \in V \,:\, \forall \sigma \in Re_1(s).\, \forall \pi \in Re_2(s).\;\; \Omega_{\sigma,\pi}(s) \models (\neg\varphi_1 \vee \neg\varphi_2) \,\}$$

Then $T_1 = T_2$.

Proof. The inclusion $T_1 \subseteq T_2$ follows from the fact that $(\neg\varphi_1 \wedge \neg\varphi_2) \rightarrow (\neg\varphi_1 \vee \neg\varphi_2)$. We show that $T_2 \subseteq T_1$. By the definition of retaliating strategies, if σ is a retaliating strategy of player 1, then for all strategies π of player 2, we have $\Omega_{\sigma,\pi}(s) \models (\varphi_2 \rightarrow \varphi_1)$, and thus $\Omega_{\sigma,\pi}(s) \models (\neg\varphi_1 \rightarrow \neg\varphi_2)$. Symmetrically, if π is a retaliating strategy of player 2, then for all strategies σ of player 1, we have $\Omega_{\sigma,\pi}(s) \models (\neg\varphi_2 \rightarrow \neg\varphi_1)$. The claim follows. $\blacksquare$

It follows from Lemma 2 and Lemma 3 that $W_{00} = V \setminus (W_{01} \cup W_{10} \cup W_{11})$. It also follows from Lemma 2 that the sets W_{01}, W_{10}, and W_{11} are disjoint. This gives the following result.

Theorem 2 (State space partition). *For all 2-player graph games with Borel objectives, the four sets W_{10}, W_{01}, W_{11}, and W_{00} form a partition of the state set.*

Lemma 4. *Consider the sets S_{ij} for $i, j \in \{ 0, 1 \}$ as defined in Definition 3. The following equalities hold:*

$$S_{00} \cap S_{01} = \emptyset; \quad S_{00} \cap S_{10} = \emptyset;$$

$$S_{01} \cap S_{10} = \emptyset; \quad S_{11} \cap S_{01} = \emptyset; \quad S_{11} \cap S_{10} = \emptyset.$$

Proof. Consider a state $s \in S_{10}$ and a secure equilibrium (σ, π) at s. Since the strategy profile is secure and player 2 gets the least possible payoff, it follows that for all player 1 strategies π', the payoff for player 1 cannot decrease. Hence for all player 2 strategies π', we have $\Omega_{\sigma,\pi'}(s) \models \varphi_1$. So there is no Nash equilibrium at state s which assigns payoff 0 to player 1. Hence we have $S_{10} \cap S_{01} = \emptyset$ and $S_{10} \cap S_{00} = \emptyset$. By symmetry, $S_{01} \cap S_{00} = \emptyset$.

Consider a state $s \in S_{11}$ and a secure equilibrium (σ, π) at s. Since the strategy profile is secure, it ensures that for all player 2 strategies π', if $\Omega_{\sigma,\pi'}(s) \models \neg\varphi_1$, then $\Omega_{\sigma,\pi'} \models \neg\varphi_2$. Hence $s \notin S_{01}$. Thus we have $S_{11} \cap S_{01} = \emptyset$, and by symmetry $S_{11} \cap S_{10} = \emptyset$. ∎

Lemma 5. *The following equalities hold:*

$$MS_{00} \cap MS_{01} = \emptyset; \quad MS_{00} \cap MS_{10} = \emptyset;$$

$$MS_{01} \cap MS_{10} = \emptyset; \quad MS_{11} \cap MS_{00} = \emptyset.$$

Proof. The first three equalities follow from Lemma 4. The last equality follows from the facts that $(0,0) \preceq_1 (1,1)$ and $(0,0) \preceq_2 (1,1)$. So if $s \in MS_{11}$, then $(0,0)$ cannot be a maximal secure payoff profile at s. ∎

Lemma 6. *$W_{11} = MS_{11}$; $W_{10} = MS_{10}$; $W_{01} = MS_{01}$.*

Proof. Consider a state $s \in MS_{10}$ and a secure equilibrium (σ, π) at s. Since player 2 gets the least possible payoff and (σ, π) is a secure strategy profile, it follows that for all strategies π' of player 2, we have $\Omega_{\sigma,\pi'}(s) \models \varphi_1$. Since (σ, π) is a Nash equilibrium, for all strategies π' of player 2, we have $\Omega_{\sigma,\pi'}(s) \models \neg\varphi_2$. Thus we have $MS_{10} \subseteq W_{10}$. Now consider a state $s \in W_{10}$ and let σ be a strongly winning strategy of player 1 at s, that is, for all strategies π of player 2, we have $\Omega_{\sigma,\pi}(s) \models (\varphi_1 \wedge \neg\varphi_2)$. For all strategies π of player 2, the strategy profile (σ, π) is a secure equilibrium. Hence, $s \in S_{10}$. Since $(1,0)$ is the greatest payoff profile in the preference ordering of the payoff profiles for player 1, we have $s \in MS_{10}$. Therefore $W_{10} = MS_{10}$. Symmetrically, $W_{01} = MS_{01}$.

Consider a state $s \in MS_{11}$ and let (σ, π) be a secure equilibrium at s. We prove that $\sigma \in Re_1(s)$ and $\pi \in Re_2(s)$. Since (σ, π) is a secure strategy profile, for all strategies π' of player 2, if $\Omega_{\sigma,\pi'}(s) \models \neg\varphi_1$, then $\Omega_{\sigma,\pi'}(s) \models \neg\varphi_2$. In other words, for all strategies π' of player 2, we have $\Omega_{\sigma,\pi'}(s) \models (\varphi_2 \rightarrow \varphi_1)$. Hence $\sigma \in Re_1(s)$. Symmetrically, $\pi \in Re_2(s)$. Thus $MS_{11} \subseteq W_{11}$. Consider a state

$s \in W_{11}$ and let $\sigma \in Re_1(s)$ and $\pi \in Re_2(s)$ such that $\Omega_{\sigma,\pi}(s) \models (\varphi_1 \wedge \varphi_2)$. A retaliation strategy profile is, by definition, a secure strategy profile. Since the strategy profile (σ, π) assigns the greatest possible payoff to each player, it is a Nash equilibrium. Therefore $W_{11} \subseteq S_{11} \subseteq MS_{11}$. ∎

Lemma 7. $W_{00} = MS_{00}$.

Proof. It follows from Lemma 4 and Lemma 5 that $MS_{00} = S_{00} \setminus S_{11} = S_{00} \setminus MS_{11}$. We will use this fact to prove that $W_{00} = MS_{00}$.

- Consider a state $s \in MS_{00}$. Then we have $s \notin (MS_{11} \cup MS_{10} \cup MS_{11}) \Rightarrow s \notin (W_{11} \cup W_{10} \cup W_{01})$. By Lemma 2 we have $W_{00}, W_{11}, W_{10}, W_{01}$ is a partition and hence we have $s \in W_{00}$. It follows that $MS_{00} \subseteq W_{00}$.
- Consider a state $s \in W_{00}$. We claim that there is a strategy σ for player 1 such that for all strategy π' we have $\Omega_{\sigma,\pi'}(s) \models \neg\varphi_2$. Assume by the way of contradiction this is not the case. By Borel determinacy then we have there is a strategy π'' for player 2 such that for all σ' we have $\Omega_{\sigma',\pi''}(s) \models \varphi_2$. It follows that either π'' is a strongly winning strategy for player 2 or a retaliation strategy such that player 2 gets payoff 1. Hence $s \notin W_{00}$, which is a contradiction. Hence there is a strategy σ such that for all π' we have $\Omega_{\sigma,\pi'}(s) \models \neg\varphi_2$. Similarly, there is a strategy π such that for all σ' we have $\Omega_{\sigma',\pi}(s) \models \neg\varphi_1$. We claim that (σ, π) is a secure equilibrium strategy profile. By property of σ for any π', $\Omega_{\sigma,\pi'}(s) \models \neg\varphi_2$. Similar argument hold for π as well. Hence we have (σ, π) is a Nash equilibrium strategy profile. For the strategy profile (σ, π) we have the payoff profile is $(0, 0)$ and it assigns the least possible payoff to each player. Hence it is a secure strategy profile. Hence $s \in S_{00}$. Also $s \in W_{00} \Rightarrow s \notin W_{11} = MS_{11}$. Hence $s \in S_{00} \setminus MS_{11}$. This gives us $W_{00} \subseteq MS_{00}$. ∎

Theorem 2 together with Lemmas 6 and 7 yields the following result.

Theorem 3 (Unique maximal secure equilibrium). *At every state of a 2-player graph game with Borel objectives, there exists a unique maximal secure equilibrium payoff profile.*

3.2 Algorithmic Characterization

We now give an alternative characterization of the sets W_{00}, W_{01}, W_{10}, and W_{11}. The new characterization is useful to derive computational complexity results for computing the four sets when player 1 and player 2 have ω-regular objectives. The characterization itself, however, is general and applies to all objectives specified as Borel sets.

Definition 7 (Cooperative strategy profiles). *Given a game graph G with state set V, and an objective $\psi \subseteq V^\omega$, we define the following sets:*

$$\langle\langle 1 \rangle\rangle_G \psi = \{s \in V \,:\, \exists \sigma \in \Sigma.\ \forall \pi \in \Pi.\ \ \Omega_{\sigma,\pi}(s) \models \psi\}$$

$$\langle\langle 2\rangle\rangle_G\, \psi = \{s \in V \,:\, \exists \pi \in \Pi.\ \forall \sigma \in \Sigma.\ \ \Omega_{\sigma,\pi}(s) \models \psi\}$$

$$\langle\langle 1,2\rangle\rangle_G\, \psi = \{s \in V \,:\, \exists \sigma \in \Sigma.\ \exists \pi \in \Pi.\ \ \Omega_{\sigma,\pi}(s) \models \psi\}$$

We omit the subscript G if it is clear from the context. Let s be a state in $\langle\langle 1,2\rangle\rangle\psi$ and let (σ,π) be a strategy profile such that $\Omega_{\sigma,\pi}(s) \models \psi$. We refer to (σ,π) as a cooperative *strategy profile at s, and informally say that the two players are cooperating to satisfy ψ.* ∎

It follows from the definitions that $W_{10} = \langle\langle 1\rangle\rangle(\varphi_1 \wedge \neg\varphi_2)$ and $W_{01} = \langle\langle 2\rangle\rangle(\varphi_2 \wedge \neg\varphi_1)$. Define $A = V \setminus (W_{10} \cup W_{01})$, the set of "ambiguous" states from which neither player has a strongly winning strategy. Let $W_i = \langle\!\langle i\rangle\!\rangle\varphi_i$, for $i \in \{1,2\}$, the winning sets of the two players, and let $U_1 = W_1 \setminus W_{10}$ and $U_2 = W_2 \setminus W_{01}$, the sets of "weakly winning" states for players 1 and 2, respectively. Define $U = U_1 \cup U_2$. Note that $U \subseteq A$.

Lemma 8. *$U \subseteq W_{11}$.*

Proof. Let $s \in U_1$. By the definition of U_1, player 1 has a strategy σ from the state s to satisfy the objective φ_1, which is obviously a retaliating strategy, because φ_1 implies $\varphi_2 \rightarrow \varphi_1$. Again by the definition of U_1, we have $s \notin W_{10}$. Hence, by the determinacy of zero-sum games (Theorem 1), player 2 has a strategy π to satisfy the objective $\neg(\varphi_1 \wedge \neg\varphi_2)$, which is a retaliating strategy, because $\neg(\varphi_1 \wedge \neg\varphi_2)$ is equivalent to $\varphi_1 \rightarrow \varphi_2$. Clearly we have $\Omega_{\sigma,\pi}(s) \models \varphi_1$ and $\Omega_{\sigma,\pi}(s) \models (\varphi_1 \rightarrow \varphi_2)$, and hence $\Omega_{\sigma,\pi}(s) \models (\varphi_1 \wedge \varphi_2)$. The case of $s \in U_2$ is symmetric. ∎

Example 2 shows that in general we have $U \subsetneq W_{11}$. Given a game graph $G = ((V,E),(V_1,V_2))$ and a subset $V' \subseteq V$ of the states, we write $G \restriction V'$ to denote the subgraph induced by V', that is, $G \restriction V' = ((V', E \cap (V' \times V')), (V_1 \cap V', V_2 \cap V'))$. The following lemma characterizes the set W_{11}.

Lemma 9. *$W_{11} = \langle\langle 1,2\rangle\rangle_{G\restriction A}(\varphi_1 \wedge \varphi_2)$.*

Proof. Let $s \in \langle\langle 1,2\rangle\rangle_{G\restriction A}(\varphi_1 \wedge \varphi_2)$. The case $s \in U$ is covered by Lemma 8, so let $s \in A \setminus U$. Let (σ,π) be a cooperative strategy profile at the state s, that is, $\Omega_{\sigma,\pi}(s) \models (\varphi_1 \wedge \varphi_2)$. Observe that if $t \in A \setminus U$ then $t \notin \langle\langle 1\rangle\rangle_G(\varphi_1)$ and $t \notin \langle\langle 2\rangle\rangle_G(\varphi_2)$. Hence, by the determinacy of the zero-sum games, from every state $t \in A \setminus U$, player 1 (resp. player 2) has a strategy $\overline{\sigma}$ (resp. $\overline{\pi}$) to satisfy the objective $\neg\varphi_2$ (resp. $\neg\varphi_1$) from the state s. We define a pair $(\sigma + \overline{\sigma}, \pi + \overline{\pi})$ of strategies from s as follows. Let $x \in A^*$ be a prefix of a play.

- When the play reaches a state $t \in U$, the players follow their winning retaliating strategies from t. It follows from Lemma 8 that $U \subseteq W_{11}$.
- If $x \in (A \setminus U)^*$, that is, if the play has not yet reached the set U, then player 1 uses the strategy σ and player 2 uses the strategy π. If, however, player 2 deviates from the strategy π, then player 1 switches to the strategy $\overline{\sigma}$ from the first state after the deviation, and symmetrically, if player 1 deviates from σ, then player 2 switches to the $\overline{\pi}$.

It is easy to observe that both strategies $(\sigma + \overline{\sigma})$ and $(\pi + \overline{\pi})$ are retaliating strategies and $\Omega_{\sigma+\overline{\sigma},\pi+\overline{\pi}}(s) \models (\varphi_1 \wedge \varphi_2)$, because $\Omega_{\sigma+\overline{\sigma},\pi+\overline{\pi}}(s) = \Omega_{\sigma,\pi}(s)$. Hence $s \in W_{11}$.

Let $s \notin \langle\langle 1,2\rangle\rangle_{G\restriction A}(\varphi_1 \wedge \varphi_2)$. Then $s \notin W_{11}$, because for every strategy profile (σ, π) we have either $\Omega_{\sigma,\pi}(s) \models \neg\varphi_1$ or $\Omega_{\sigma,\pi}(s) \models \neg\varphi_2$. ∎

We now define two forms of ω-regular objectives, Rabin and parity objectives. For an infinite path $\Omega = \langle s_0, s_1, s_2, \ldots\rangle$, we define $\mathrm{Inf}(\Omega) = \{\, s \in V : s_k = s \text{ for infinitely many } k \geq 0 \,\}$.

- *Rabin:* We are given a set $\alpha \subseteq 2^V \times 2^V$ of pairs such that $\alpha = \{\, (E_1, F_1), (E_2, F_2), \ldots, (E_d, F_d) \,\}$, where $E_i, F_i \subseteq V$ for all $1 \leq i \leq d$. A *Rabin objective* has the form $\varphi_{Rabin} = \{\, \Omega \in V^\omega : \text{there exists } 1 \leq i \leq d \text{ such that } \mathrm{Inf}(\Omega) \cap E_i = \emptyset \text{ and } \mathrm{Inf}(\Omega) \cap F_i \neq \emptyset \,\}$.
- *Parity:* For $d \in \mathbb{N}$, we write $[d]$ to denote the set $\{0, 1, \ldots, d\}$, and $[d]_+ = \{1, 2, \ldots, d\}$. We are given a function $p\colon V \to [d]$ that assigns a *priority* $p(s)$ to every state $s \in V$. A *parity (or Rabin chain) objective* has the form $\varphi_P = \{\, \Omega \in V^\omega : \min\big(p(\mathrm{Inf}(\Omega))\big) \text{ is even} \,\}$.

Every ω-regular set can be defined as a parity objective [17]. It follows from Lemma 9 that in order to compute the sets W_{10}, W_{01}, W_{11}, and W_{00}, it suffices to solve two games with conjunctive objectives and a model-checking (1-player) problem for a conjunctive objective. If the objectives φ_1 and φ_2 are ω-regular sets specified as parity objectives, then the conjunctions can be expressed as the complement of a Rabin objective [17]. This gives the following result. (The *size* of a game graph G is $|V| + |E|$).

Theorem 4 (Complexity of computing secure equilibria). *Consider a game graph G of size n, and two Borel objectives φ_1 and φ_2 for the two players.*

- *The four sets W_{10}, W_{01}, W_{11}, and W_{00} can be computed as $W_{10} = \langle\langle 1\rangle\rangle_G(\varphi_1 \wedge \neg\varphi_2)$; $W_{01} = \langle\langle 2\rangle\rangle_G(\varphi_2 \wedge \neg\varphi_1)$; $W_{11} = \langle\langle 1,2\rangle\rangle_{G\restriction A}(\varphi_1 \wedge \varphi_2)$, where $A = V \setminus (W_{10} \cup W_{01})$; and $W_{00} = V \setminus (W_{10} \cup W_{01} \cup W_{11})$.*
- *If φ_1 and φ_2 are ω-regular objectives specified as LTL formulas, then deciding W_{10}, W_{01}, W_{11}, and W_{00} is 2EXPTIME-complete. The four sets can be computed in time $O\big(n^{2^\ell} \times 2^{2^{\ell \log \ell}}\big)$, where $\ell = |\varphi_1| + |\varphi_2|$ [15].*
- *If φ_1 and φ_2 are parity objectives, then W_{10}, W_{01}, W_{11}, and W_{00} can be decided in co-NP. The four sets can be computed in time $O\big((nd)^{2d}\big)$, where d is the maximal number of priorities in the priority functions for φ_1 and φ_2 [5,4].*

4 ω-Regular Objectives

In this section we consider special cases of graph games, where the two players have reachability, safety, Büchi, co-Büchi, and parity objectives. We fix a game graph G with state space V. Given state sets $R, S, B, C \subseteq V$, these objectives are defined as follows.

1. *Reachability:* $\varphi_R = \{s_0 s_1 \ldots \in V^\omega : \exists k.\ s_k \in R\}$. We refer to R as the *target* set.
2. *Safety:* $\varphi_S = \{s_0 s_1 \ldots \in V^\omega : \forall k.\ s_k \in S\}$. We refer to S as the *safe* set.
3. *Büchi:* $\varphi_B = \{s_0 s_1 \ldots \in V^\omega : \forall k.\ \exists l > k.\ s_l \in B\}$. We refer to B as the *Büchi* set.
4. *co-Büchi:* $\varphi_C = \{s_0 s_1 \ldots \in V^\omega : \exists k.\ \forall l > k.\ s_l \in C\}$. We refer to C as the *co-Büchi* set.

Parity objectives were defined in the previous section. Note that Büchi and co-Büchi objectives are special cases of parity objectives with two priorities: in the Büchi case, take the priority function $p\colon V \to [1]$ such that $p(s) = 0$ if $s \in B$, and $p(s) = 1$ otherwise; in the co-Büchi case, take the priority function $p\colon V \to [2]_+$ such that $p(s) = 2$ if $s \in C$, and $p(s) = 1$ otherwise.

We characterize the memory requirements for strongly winning and retaliating strategies if both players have ω-regular objectives. A retaliation strategy profile (σ, π) is called *winning* at a state $s \in V$ if $\Omega_{\sigma,\pi}(s) \models (\varphi_1 \wedge \varphi_2)$. A strategy σ is a *winning* retaliating strategy for player 1 at state s if there is a strategy π for player 2 such that (σ, π) is a winning retaliation strategy profile at s. Until the end of this section, let φ_R be a reachability objective, φ_S a safety objective, φ_B a Büchi objective, φ_C a co-Büchi objective, and φ_P a parity objective.

Proposition 1 (Conjunctive objectives as parity objectives).

1. *$\neg\varphi_R$ is a safety objective and $\neg\varphi_S$ is a reachability objective,*
2. *$\neg\varphi_C$ is a Büchi objective, and $\neg\varphi_B$ is a co-Büchi objective.*
3. *$\neg\varphi_P$, $\varphi_S \wedge \varphi_P$, and $\varphi_C \wedge \varphi_P$ are parity objectives.*

Proof. A negation of a parity objective with priority function p can be obtained as the parity objective with the priority function $p'(s) = p(s) + 1$. It follows that the negation of a Büchi objective is equivalent to a co-Büchi objective and the negation of a co-Büchi objective is equivalent to a Büchi objective.

If φ_P is a parity objective and φ_D is a safety objective or a co-Büchi objective then the conjunction $\varphi_D \wedge \varphi_P$ is equivalent to a parity objective. For example, the conjunction of a parity objective φ_P and a coBüchi objective φ_D is a parity objective with the following priority function:

$$p'(s) = \begin{cases} 1 & \text{if } s \notin D, \\ p(s) + 2 & \text{if } s \in D. \end{cases}$$

The result for conjunction of parity and safety objective follows from similar construction. ∎

While in zero-sum games played on graphs, memoryless winning strategies exists for all parity objectives [6], this is not the case for non-zero-sum games. The following two theorems give a complete characterization.

Theorem 5. *If player 1 has a strongly winning strategy in a graph game where both players have reachability, safety, Büchi, co-Büchi, or parity objectives φ_1 and φ_2, then player 1 has a memoryless strongly winning strategy if and only if there is a "+" symbol in the corresponding entry of the Table 1.*

Table 1. Strongly winning strategies

		φ_2				
		φ_R	φ_B	φ_C	φ_P	φ_S
φ_1	φ_S	+	+	+	+	+
	φ_C	+	+	+	+	−
	φ_B	+	+	−	−	−
	φ_P	+	+	−	−	−
	φ_R	+	−	−	−	−

Table 2. Winning retaliating strategies

		φ_2				
		φ_R	φ_B	φ_C	φ_P	φ_S
φ_1	φ_S	+	+	+	+	+
	φ_C	+	−	−	−	−
	φ_B	+	−	−	−	−
	φ_P	+	−	−	−	−
	φ_R	+	−	−	−	−

Proof. For player 1, strongly winning a non-zero-sum game with objectives φ_1 and φ_2 is equivalent to winning a zero-sum game with the objective $\varphi_1 \wedge \neg\varphi_2$. Hence by existence of memoryless winning strategies for zero-sum parity games [6] player 1 has memoryless strongly winning strategies if the objective $\varphi_1 \wedge \neg\varphi_2$ is equivalent to a parity objective. Using Proposition 1 it is easy to observe that the objective $\varphi_1 \wedge \neg\varphi_2$ is equivalent to a parity objective for all "+" entries in Table 1, except for safety–reachability, safety–safety, and reachability–reachability games. For these three cases, it is easy to argue that memoryless strongly winning strategies exist. The other "+" entries follow from the existence of memoryless winning strategies for zero-sum parity games [6].

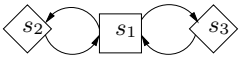


Fig. 3. A counterexample for memoryless strongly winning strategies

We now show that player 1 does not necessarily have a memoryless strongly winning strategy in non-zero-sum games with "−" entries in Table 1. It suffices to give counterexamples for the following four cases: co-Büchi–safety, Büchi–safety, reachability–safety, and Büchi–co-Büchi games. The cases of reachability–Büchi and reachability–co-Büchi games follow from the former two cases, respectively, by symmetry. The cases of Büchi–parity and parity–parity games follow trivially from the Büchi–co-Büchi case, and the case of parity–safety games follows trivially from the Büchi–safety case. The game graph of Fig. 3 serves as a

counterexample for all four cases. For all the cases, let $C = S = \{s_1, s_2\}$ and $B = R = \{s_2\}$.

For the co-Büchi–safety case, the player 1 strategy that chooses $s_1 \rightarrow s_3$ for the first time and then always chooses $s_1 \rightarrow s_2$ is strongly winning at the state s_1, but the two possible memoryless strategies are not strongly winning. For all other cases, the player 1 strategy that alternates between the two moves available at s_1 is strongly winning, but again the two memoryless strategies are not. ∎

Theorem 6. *If player 1 has a winning retaliating strategy in a graph game where both players have reachability, safety, Büchi, co-Büchi, or parity objectives φ_1 and φ_2, then player 1 has a memoryless winning retaliating strategy if and only if there is a "+" symbol in the corresponding entry of the Table 2.*

Proof. First we show that player 1 has memoryless winning retaliating strategies in parity–reachability and safety–parity games. Recall the weakly winning sets $U_1 = W_1 \setminus W_{10}$ and $U_2 = W_2 \setminus W_{01}$, where $W_i = \langle\langle i \rangle\rangle \varphi_i$ for $i \in \{1, 2\}$. In $U_1 \subseteq W_{11}$ player 1 uses her memoryless winning strategy in the zero-sum game with the objective φ_P. In $W_{11} \setminus U_1$ player 1 uses a memoryless strategy that shortens the distance in the game graph to the set U_1. This strategy is a winning retaliating strategy for player 1 in U_1, because it satisfies the objective φ_P. We prove that it is also a winning retaliating strategy for player 1 in $W_{11} \setminus U_1$, that is, satisfaction of the objective φ_R implies satisfaction of the objective φ_P. Observe that $R \cap (W_{11} \setminus U_1) = \emptyset$. Otherwise there would be a state in $W_{11} \setminus U_1$ in which the objective φ_R of player 2 is satisfied and player 2 has a strategy to satisfy $\neg\varphi_P$, and hence the state belongs to W_{01}; this however contradicts $W_{11} \cap W_{01} = \emptyset$. Therefore, as long as a play stays in $W_{11} \setminus U_1$, the objective φ_R cannot be satisfied. On the other hand, if player 2 cooperates with player 1 in reaching U_1, then player 1 plays her memoryless retaliating strategy in U_1. The proof for safety–parity games is similar. There, the key observation is that $W_{11} \setminus U_1 \subseteq S$, where φ_S is the safety objective of player 1.

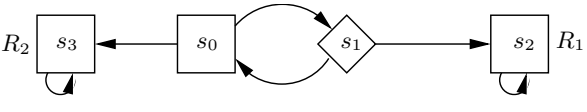


Fig. 4. A counterexample for memoryless winning retaliating strategies

We now argue that player 1 does not have memoryless winning retaliating strategies in games with "−" entries in Table 2. It suffices to give counterexamples for the nine cases that result from co-Büchi, Büchi, or reachability objectives for player 1, and Büchi, co-Büchi, or safety objectives for player 2. The remaining seven cases involving parity objectives follow as corollaries, because Büchi and co-Büchi objectives are special cases of parity objectives. The game graph of Fig. 4 serves as a counterexample for all nine cases: take $C_1 = B_1 = R_1 = \{s_2\}$

and $B_2 = C_2 = S_2 = \{s_0, s_1, s_2\}$, where C_1, B_1, and R_1 are the co-Büchi, Büchi, and reachability objectives of player 1, respectively, and B_2, C_2, and S_2 are the Büchi, co-Büchi, and safety objectives of player 2. It can be verified that in each of the nine games neither of the two memoryless strategies for player 1 is a winning retaliating strategy at the state s_0, but the strategy that first chooses the move $s_0 \to s_1$ and then chooses $s_0 \to s_3$ if player 2 chooses $s_1 \to s_0$, is a winning retaliating strategy for player 1. ∎

Note that if both players have parity objectives, then at all states in W_{00} memoryless retaliation strategy profiles exist. To see this, consider a state $s \in W_{00}$. There are a player 1 strategy $\overline{\sigma}$ and a player 2 strategy $\overline{\pi}$ such that for all strategies σ of player 1 and π of player 2, we have $\Omega_{\sigma,\overline{\pi}}(s) \models \neg\varphi_1$ and $\Omega_{\overline{\sigma},\pi}(s) \models \neg\varphi_2$. The strategy profile $(\overline{\sigma}, \overline{\pi})$ is a retaliation strategy profile. If the objectives φ_1 and φ_2 are both parity objectives, then $\neg\varphi_1$ and $\neg\varphi_2$ are parity objectives as well. Hence there are memoryless strategies $\overline{\sigma}$ and $\overline{\pi}$ that satisfy the above condition.

5 *n*-Player Games

We generalize the definition of secure equilibria to the case of $n > 2$ players. We show that in n-player games on graphs, in contrast to the 2-player case, there may not be a unique maximal secure equilibrium. The preference ordering $\prec_i$ for player i, where $i \in \{1, \ldots, n\}$, is defined as follows: given two payoff profiles $v = (v_1, \ldots, v_n)$ and $v' = (v'_1, \ldots, v'_n)$, we have $v \prec_i v'$ iff $(v'_i > v_i) \vee (v'_i = v_i \wedge (\forall j \neq i.\ v'_j \leq v_j) \wedge (\exists j \neq i.\ v'_j < v_j))$. In other words, player i prefers v' over v iff she gets a greater payoff in v', or (1) she gets equal payoff in v' and v, (2) the payoff of every other player is no more in v' than in v, and (3) there is at least one player who gets a lower payoff in v' than in v. Given a strategy profile $\sigma = (\sigma_1, \ldots, \sigma_n)$, we define the corresponding payoff profile as $v^\sigma = (v_1^\sigma, \ldots, v_n^\sigma)$, where v_i^σ is the payoff for player i when all players choose their strategies from the strategy profile σ. Given a strategy σ'_i for player i, we write (σ_{-i}, σ'_i) for the strategy profile where each player $j \neq i$ plays the strategy σ_j, and player i plays the strategy σ'_i. An n-player strategy profile σ is *Nash equilibrium* if for all players i and all strategies σ'_i of player i, if $\sigma' = (\sigma_{-i}, \sigma'_i)$, then $v_i^{\sigma'} \leq v_i^\sigma$.

Definition 8 (Secure *n*-player profile). *An n-player strategy profile σ is* secure *if for all players i and $j \neq i$, and for all strategies σ'_j of player j, if $\sigma' = (\sigma_{-j}, \sigma'_j)$, then $(v_j^{\sigma'} \geq v_j^\sigma) \to (v_i^{\sigma'} \geq v_i^\sigma)$.* ∎

Observe that if a secure profile σ is interpreted as a contract between the players, then any unilateral selfish deviation from σ must be cooperative in the following sense: if player j deviates from the contract σ by playing a strategy σ'_j (i.e., the new strategy profile is $\sigma' = (\sigma_{-j}, \sigma'_j)$) which gives her an advantage (i.e., $v_j^{\sigma'} \geq v_j^\sigma$), then every other player $i \neq j$ is not put at a disadvantage if she follows the contract (i.e., $v_i^{\sigma'} \geq v_i^\sigma$). By symmetry, the player j enjoys the same security against unilateral selfish deviations of other players.

Definition 9 (Secure n-player equilibrium). *A n-player strategy profile σ is a* secure equilibrium *if σ is both a Nash equilibrium and secure.* ∎

Similar to Lemma 1 we have the following result.

Lemma 10 (Equivalent characterization). *An n-player strategy profile σ is a* secure equilibrium *iff for all players i, there does not exist a strategy σ_i' of player i such that $\sigma' = (\sigma_{-i}, \sigma_i')$ and $v^\sigma \prec_i v^{\sigma'}$.*

We give an example of a 3-player graph game where the maximal secure equilibrium payoff profile is not unique. Recall the game graph from Fig. 3, and consider a 3-player game on this graph where each player has a reachability objective. The target set for player 1 is $\{s_2, s_3\}$; for player 2 it is $\{s_2\}$; and for player 3 it is $\{s_3\}$. In state s_1 player 1 can chose between the two successors s_2 and s_3. If player 1 chooses $s_1 \to s_3$, then the payoff profile is $(1, 0, 1)$, and if player 1 chooses $s_1 \to s_2$, then the payoff profile is $(1, 1, 0)$. Both are secure equilibria and maximal, but incomparable.

6 Conclusion

We considered non-zero-sum graph games with lexicographically ordered objectives for the players in order to capture adversarial external choice, where each player tries to minimize the other player's payoff as long as it does not decrease her own payoff. We showed that these games have a unique maximal equilibrium for all Borel winning conditions. This confirms that secure equilibria provide a good formalization of rational behavior in the context of verifying component-based systems.

Concretely, suppose the two players represent two components of a system with the specifications φ_1 and φ_2, respectively. Classically, component-wise verification would prove that for an initial state s, player 1 can satisfy the objective φ_1 no matter what player 2 does (i.e., $s \in \langle\langle 1 \rangle\rangle \varphi_1$), and player 2 can satisfy the objective φ_2 no matter what player 1 does (i.e., $s \in \langle\langle 2 \rangle\rangle \varphi_2$). Together, these two proof obligations imply that the composite system satisfies both specifications φ_1 and φ_2. The computational gain from this method typically arises from abstracting the opposing player's (i.e., the environment's) moves for each proof obligation. Our framework provides two weaker proof obligations that support the same conclusion. We first show that player 1 can satisfy φ_1 provided that player 2 does not sabotage her ability to satisfy φ_2, that is, we show that $s \in (W_{10} \cup W_{11})$: either player 1 has a strongly winning strategy, or there is a winning pair of retaliation strategies. This condition is strictly weaker than the condition that player 1 has a winning strategy, and therefore it is satisfied by more states. Second, we show the symmetric proof obligation that player 2 can satisfy φ_2 provided that player 1 does not sabotage her ability to satisfy φ_1, that is, $s \in (W_{01} \cup W_{11})$. While they are weaker than their classical counterparts, both new proof obligations together still suffice to establish that $s \in W_{11}$, that is, the composite system satisfies $\varphi_1 \wedge \varphi_2$ assuming that both players behave rationally and follow the winning pair of retaliation strategies.

It should be noted that the other possible lexicographic ordering of objectives captures *cooperative* external choice, where each player tries to *maximize* the other player's payoff as long as it does not decrease her own payoff. However, cooperation does not uniquely determine a preferable behavior: there may be multiple maximal payoff profiles for cooperative external choice, even for reachability objectives. To see this, define $(v_1, v_2) \prec_1^{co} (v_1', v_2')$ iff $(v_1 < v_1') \vee (v_1 = v_1' \wedge v_2 < v_2')$, and $(v_1, v_2) \preceq_1^{co} (v_1, , v_2')$ iff $(v_1, v_2) \prec_1^{co} (v_1', v_2')$ or $(v_1, v_2) = (v_1', v_2')$. A symmetric definition yields $\preceq_2^{co}$. A *cooperative equilibrium* is a Nash equilibrium with respect to the precedence orderings $\preceq_1^{co}$ and $\preceq_2^{co}$ on payoff profiles. Consider the game shown in Fig. 4, where each player has a reachability objective. The target for player 1 is s_2, and the target for player 2 is s_3. The possible cooperative equilibria at state s_0 are as follows: player 1 chooses $s_0 \to s_1$ and player 2 chooses $s_1 \to s_2$, or player 1 chooses $s_0 \to s_3$ and player 2 chooses $s_1 \to s_0$. The former equilibrium has the payoff profile $(1, 0)$, and the latter has the payoff profile $(0, 1)$. These are the only cooperative equilibria and, therefore, the maximal payoff profile for cooperative equilibria is not unique.

Acknowledgment. We thank Christos Papadimitriou for helpful discussions regarding the formalization of rational behavior in game theory.

References

1. M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17:507–534, 1995.
2. R. Alur and T.A. Henzinger. Reactive modules. In *Formal Methods in System Design*, 15:7–48, 1999.
3. R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49:672–713, 2002.
4. S. Dziembowski, M. Jurdziński, and I. Walukiewicz. How much memory is needed to win infinite games? In *Logic in Computer Science* (LICS), pages 99–110. IEEE Computer Society Press, 1997.
5. E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Foundations of Computer Science* (FOCS), pages 328–337. IEEE Computer Society Press, 1988.
6. E.A. Emerson and C. Jutla. Tree automata, μ-calculus, and determinacy. In *Foundations of Computer Science* (FOCS), pages 368–377. IEEE Computer Society Press, 1991.
7. J.F. Nash Jr. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences*, 36:48–49, 1950.
8. A. Kechris. *Classical Descriptive Set Theory.* Springer-Verlag, 1995.
9. D.M. Kreps. *A Course in Microeconomic Theory.* Princeton University Press, 1990.
10. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.
11. D.A. Martin. Borel determinacy. *Annals of Mathematics*, 102:363–371, 1975.
12. D.A. Martin. The determinacy of Blackwell games. *Journal of Symbolic Logic*, 63:1565–1581, 1998.

13. K. Namjoshi N. Amla, E.A. Emerson and R. Trefler. Abstract patterns for compositional reasoning. In *Concurrency Theory* (CONCUR), LNCS 2761, pages 423–448. Springer-Verlag, 2003.
14. G. Owen. *Game Theory.* Academic Press, 1995.
15. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Principles of Programming Languages* (POPL), pages 179–190. ACM Press, 1989.
16. W. Thomas. On the synthesis of strategies in infinite games. In *Symposium on Theoretical Aspects of Computer Science* (STACS), LNCS 900, pages 1–13. Springer-Verlag, 1995.
17. W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, eds., *Handbook of Formal Languages*, volume 3, pages 389–455. Springer-Verlag, 1997.

Priced Timed Automata: Algorithms and Applications

Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen[⋆,⋆⋆]

Aalborg University

Abstract. This contribution reports on the considerable effort made recently towards extending and applying well-established timed automata technology to optimal scheduling and planning problems. The effort of the authors in this direction has to a large extent been carried out as part of the European projects VHS [22] and AMETIST [17] and are available in the recently released UPPAAL CORA [12], a variant of the real-time verification tool UPPAAL [20,5] specialized for cost-optimal reachability for the extended model of priced timed automata.

1 Introduction and Motivation

Since its introduction by Alur and Dill [2] the model of timed automata has established itself as a standard modeling formalism for describing real-time system behavior. A number of mature model checking tools (e.g. KRONOS, UPPAAL, IF [11,20,16]) are by now available and have been applied to the quantitative analysis of numerous industrial case-studies [25].

An interesting application of real-time model checking that has recently been receiving substantial attention is to extend and re-target the timed automata technology towards optimal scheduling and planning. The extensions include most importantly an augmentation of the basic timed automata formalism allowing for the specification of the accumulation of cost during behavior [7,3]. The state-exploring algorithms have been modified to allow for "guiding" the (symbolic) state-space exploration in order that "promising" and "cheap" states are visited first, and to apply branch-and-bound techniques [6] to prune parts of the search tree that are guaranteed not to improve on solutions found so far. Also new symbolic data structures allowing for efficient symbolic state-space representation with additional cost-information have been introduced and implemented in order to efficiently obtain optimal or near-optimal solutions [19]. Within the VHS and AMETIST projects successful applications of this technology have been made to a number of benchmark examples and industrial case studies. With this new direction, we are entering the area of Operations Research and Artificial Intelligence with a well-established and extensive list of existing techniques (MILP, constraint programming, genetic programming, etc.). However, what we put forward is a completely new and promising technology based

⋆ BRICS, Aalborg University, Denmark.

⋆⋆ Work partially done within the European IST project AMETIST.

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 162–182, 2005.
© Springer-Verlag Berlin Heidelberg 2005

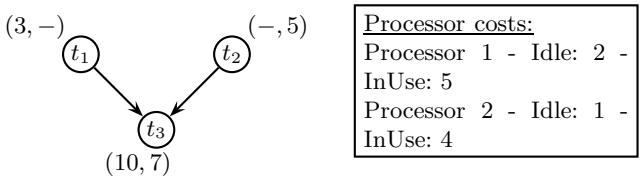


Fig. 1. Task graph scheduling problem with 3 tasks and 2 processors

on the efficient algorithms/data structures coming from timed automata analysis, and allowing for very natural and compositional descriptions of even highly non-standard scheduling problems with timing constraints.

Abstractly, a scheduling or planning problem may be understood in terms of a number of *objects* (e.g. a number of different cars, persons) each associated with various distinguishing attributes (e.g. speed, position). The possible plans solving the problem are described by a number of *actions*, the execution of which may depend on and affect the values of (some of) the objects attributes. Solutions, or feasible schedules, come in (at least) two flavors:

Finite Schedule: a finite sequence of actions that takes the system from the initial configuration to one of a designated collection of desired goal configurations.

Infinite Schedule: an infinite sequence of actions that – when starting in the initial configuration – ensures that the system configuration stays indefinitely within a designated collection of desired configurations.

In order to reinforce quantitative aspects, actions may additionally be equipped with constraints on durations and have associated costs. In this way one may distinguish different feasible schedules according to their accumulated cost or time (for finite schedules) or their cost per time ratio in the limit (for infinite schedules) in identifying *optimal* schedules. It is understood that independent actions, in terms of the set of objects the actions depend upon and affect, may overlap time-wise.

One concrete scheduling problem is that of optimal *task graph scheduling* (TGS) consisting in scheduling a number of interdependent tasks (e.g. performing some arithmetic operations) onto a number of heterogeneous processors. The interdependencies state that a task cannot start executing before all its predecessors have terminated. Furthermore, each task can only execute on a subset of the processors. An example task graph with three tasks is depicted in Fig. 1. The task t_3 cannot start executing until both tasks t_1 and t_2 have terminated. The available resources are two processors p_1 and p_2. The tasks (nodes) are annotated with the required execution times on the processors, that is, t_1 can only execute on p_1, t_2 only on p_2 while t_3 can execute on both p_1 and p_2. Furthermore, the idling costs per time unit of the processors are 2 and 1, respectively, and operations costs per time unit are 5 and 4, respectively.

Now, scheduling problems are naturally modeled using networks of timed automata. Each object is modeled as a separate timed automaton annotated with local, discrete variables representing the attributes associated with the object.

Interaction often involves only a few objects and can be modeled as synchronizing edges in the timed automata models of the involved objects. Actions involving time durations are naturally modeled using guarded edges over clock variables. Furthermore, operation costs can be associated with states and edges in the model of priced timed automata (PTA) which was, independently, introduced in [7] and [3]. The separation of independent objects into individual processes and representing interaction between objects as synchronizing actions allows timed automata to make the control flow of scheduling problems explicit. In turn, this makes the models intuitively understood and easy to communicate. Figure 2 depicts PTA models for the task graph in Fig. 1 and is explained in detail in Section 4.2.

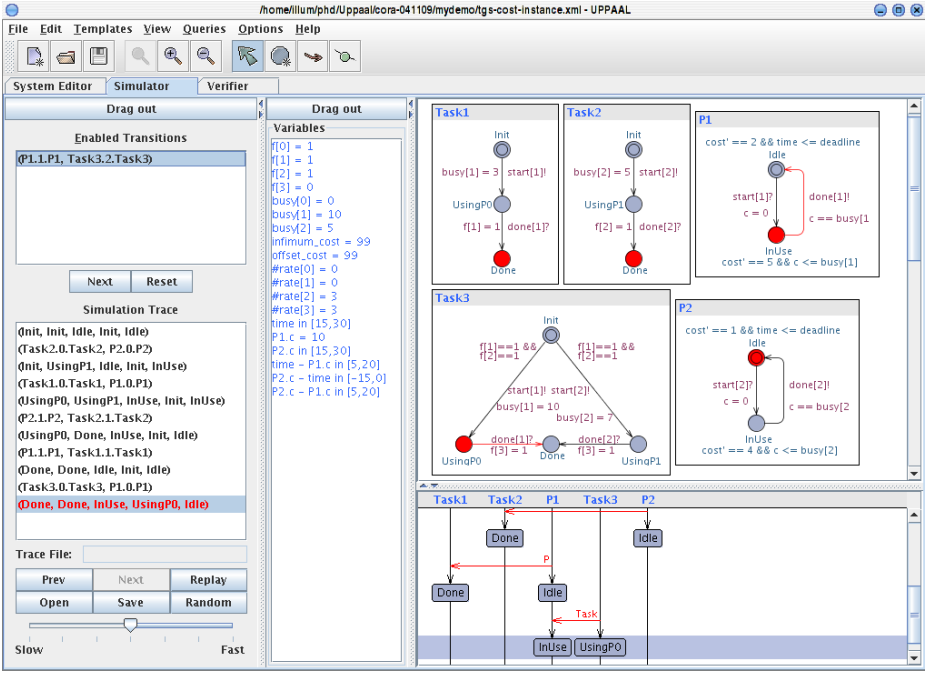


Fig. 2. Screen shot of the Uppaal Cora simulator for the task graph scheduling problem of Fig. 1

The outline of the remainder of the paper is as follows: In Sections 2 and 3 we introduce the model of PTA, the problem of cost-optimal reachability and sketch the symbolic branch-and-bound algorithm used by Uppaal Cora for solving this problem. Then in Section 4 we show how to model a range of generic scheduling problems using PTA, provide experimental results and describe two industrial scheduling case-studies. Finally, in Section 5, we comment on other PTA-related optimization problems to be supported in future releases of Uppaal Cora.

2 Priced Timed Automata

In this section we give a formal definition of priced timed automata (PTA) and their semantics[1]. Let X be a set of clocks. Intuitively, clocks are non-negative real valued variables that can be reset to zero and grow at a fixed rate with the passage of time. A priced timed automaton over X is an annotated directed graph with a distinguished vertex called the initial location. In the tradition of timed automata, we call vertices *locations*. An edge is decorated with a guard, an action and a reset set. We say that an edge is enabled if the guard evaluates to true and the source location is active. A reset set is a set of clocks. The intuition is that the clocks in the reset set are set to zero whenever the edge is taken. Note that following edges is instantaneous and thus takes no time. Finally, locations are labeled with invariants. Intuitively, an invariant must evaluate to true whenever its location is active. Both guards and invariants are conjunctions of simple constraints $x \bowtie k$, where x is a clock in X, k is a non-negative integer value, and $\bowtie \in \{<, \leq, =, \geq, >\}$. Let $\mathcal{B}(X)$ be the set of all such expressions. The previous definition is in fact that of a timed automaton. To form a priced timed automaton, we annotate the edges and the locations with costs and cost rates, respectively. The above is summarized in the following definition.

Definition 1 (Priced Timed Automata). *Let X be a set of clocks and Act a set of actions. A priced timed automaton over X and Act is a tuple $A = (L, E, l_0, I, P)$, where L is a set of locations, $E \subseteq L \times \mathcal{B}(X) \times Act \times 2^X \times L$ is a set of edges, $l_0 \in L$ is the initial location, $I : L \to \mathcal{B}(X)$ assigns invariants to locations, and $P : L \cup E \to \mathbb{N}_0$ assigns cost rates and costs to locations and edges, respectively.*

The semantics of a PTA is defined as a *priced transition system*. A priced transition system is a labeled transition system, where the transition relation is given by a partial function from transitions to the non-negative reals, intuitively being the cost of the transition. We write $s \xrightarrow{a}_p s'$ whenever the function is defined on the transition (s, a, s') and the cost is p.

Definition 2 (Priced Transition System). *A priced transition system is a tuple $\mathcal{T} = (S, s_0, \Sigma, \to)$, where S is a (possibly infinite) set of states, $s_0 \in S$ is the initial state, Σ is a set of labels, and $\to : (S \times \Sigma \times S) \hookrightarrow \mathbb{R}_{\geq 0}$ is a partial function from transitions to the non-negative reals.*

In case of PTA, a state consists of the active location $l \in L$ and a valuation of all clocks $v : X \to \mathbb{R}_{\geq 0}$ such that the invariant of l evaluates to true for v. There are two types of transitions between these states: *discrete transitions* and *delay transitions*. That is, transitions that instantaneously change the control location of the automaton without time passing and transitions that pass time in a fixed control location, respectively. Consequently, the labels of the corresponding priced transition system consists of the labels of the priced timed automaton and the non-negative reals. We formalize this in the following definition.

[1] We ignore the syntactic extensions of discrete variables and parallel composition of automata and note that these can be added easily.

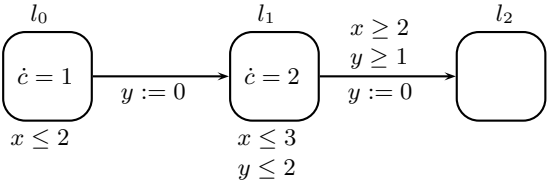


Fig. 3. A priced timed automaton, A

Definition 3 (Semantics of a Priced Timed Automaton). *The semantics of a PTA $A = (L, E, l_0, I, P)$ over clocks X and actions Act is given by a priced transition system $\mathcal{T} = (S, s_0, \Sigma, \rightarrow)$, where $S = \{(l, u) \in L \times \mathbb{R}^X_{\geq 0} \mid u \models I(l)\}$ is the set of states satisfying the invariants, $s_0 = (l_0, u_0)$ is the initial state for u_0 evaluating to zero for all clocks in X, $\Sigma = Act \cup \mathbb{R}_{\geq 0}$ is the set of labels, and $\rightarrow$ consists of discrete and delay transitions as defined below.*

Discrete transitions are the result of following an enabled edge in the PTA. As a result, the destination location is activated and the clocks in the reset set are set to zero. The cost of the transition is given by the cost of the edge.

Definition 4 (Discrete transitions). *A transition $(l, v) \xrightarrow{a}_p (l', v')$ is a discrete transition iff there is an edge (l, g, a, r, l') from l to l', such that the guard, g, evaluates to true in the source state (l, v), v' is derived from v by resetting all clocks in the reset set, r, and $p = P(e)$ is the cost of the edge.*

Delay transitions are the result of the passage of time and do not cause a change of location. A delay is only valid if the invariant of the active location is satisfied by all intermediary states. The cost of a delay transition is given by the product of the duration of the delay and the cost rate of the active location.

Definition 5 (Delay transitions). *A transition $(l, v) \xrightarrow{d}_p (l, v')$ is a delay transition iff $p = d \cdot P(l)$, $v' = v + d$,[2] and the invariant of l is satisfied by the source, target and all intermediary states, i.e., for all non-negative delays d' less than or equal to d we have $v + d' \models I(l)$.*

For networks of timed automata we use vectors of locations and the cost rate of a vector of locations is the sum of cost rates in the locations of the vector.

Example 1. Now consider the priced timed automaton A in Fig. 3 having two clocks x and y, a single goal location l_2 and two locations l_0 and l_1 with cost rate 1 and 2 respectively. Below we offer three sample traces of A:

$$\alpha_0 = (l_0, x = 0, y = 0) \rightarrow_0 (l_1, x = 0, y = 0) \xrightarrow{2}_4 (l_1, x = 2, y = 2) \rightarrow_0 (l_2, x = 2, y = 0)$$

[2] $v + d$ is the clock valuation derived from v by incrementing all clocks by d.

$$\alpha_1 = (l_0, x = 0, y = 0) \xrightarrow{2}_2 (l_0, x = 2, y = 2) \rightarrow_0 (l_1, x = 2, y = 0)$$
$$\xrightarrow{1}_2 (l_1, x = 3, y = 1) \rightarrow_0 (l_2, x = 3, y = 0)$$

$$\alpha_2 = (l_0, x = 0, y = 0) \xrightarrow{1}_1 (l_0, x = 1, y = 1) \rightarrow_0 (l_1, x = 1, y = 0)$$
$$\xrightarrow{1}_2 (1_1, x = 2, y = 1) \rightarrow_0 (l_2, x = 2, y = 0) \quad \square$$

3 Optimal Scheduling

We now turn to the definition of the optimal reachability problem for PTA and provide a brief and intuitive overview of Uppaal Cora's branch and bound algorithm for cost-optimal reachability analysis.

Cost-optimal reachability is the problem of finding the minimum cost of reaching a given goal location. More formally, an execution of a PTA is a path in the priced transition system defined by the PTA (see above), i.e., $\alpha = s_0 \xrightarrow{a_1}_{p_1} s_1 \xrightarrow{a_2}_{p_2} s_2 \cdots \xrightarrow{a_n}_{p_n} s_n$. The cost, $cost(\alpha)$, of execution α is the sum of all the costs along the execution, i.e. $\sum_i p_i$. The minimum cost, $mincost(s)$ of reaching a state s is the infimum of the costs of all finite executions from s_0 to s. Given a PTA with location l, the *cost-optimal reachability problem* is to find the largest cost k such that $k \leq mincost((l, v))$ for all clock valuations v.

Example 2. Referring to example 1, the accumulated cost of the three traces are, respectively, $cost(\alpha_0) = cost(\alpha_1) = 4$ and $cost(\alpha_2) = 3$. Thus, among the three suggested traces, α_2, leads to l_2 with minimum cost. In fact, as we shall see later, this is the minimum cost by which l_2 may be reached by *any* trace of A. $\square$

Since clocks are defined over the non-negative reals, the priced transition system generated by a PTA can be uncountably infinite, thus an enumerative, explicit state approach to the cost-optimal reachability problem is infeasible. Instead, we build upon the work done for timed automata by using *priced symbolic states*. Priced symbolic states provide symbolic representations of possibly infinite sets of actual states and their association with costs. The idea is that during exploration, the infimum cost along a symbolic path (a path of symbolic states) is stored in the symbolic state itself. If the same state is reached with different costs along different paths, the symbolic states can be compared, discarding the more expensive state.

Analogous to timed automata, a priced symbolic state of a PTA can be represented as a location and a priced zone. Priced zones describe sets of clock valuations and their associated costs. The set of clock valuations is described as a simple constraint system over clocks and differences between clocks, called zones. The cost is an affine hyperplane in an $|X| + 1$ dimensional Euclidean space, where each point is a clock valuation and an associated cost, i.e., for each clock valuation in the zone, the hyperplane provides a cost of that valuation. We observe that the constraint system describing a zone can be simplified by adding an additional clock, $\mathbf{0}$, that by definition is zero in all valuations. Then constraints on individual clocks can be represented as constraints on differences

between clocks, e.g., $x < 5$ becomes $x - \mathbf{0} < 5$. A zone be efficiently represented as a *difference bound matrix* or DBM [13]. It represents the constraint system describing the zone as a $|X| + 1$ dimensional matrix, with entries c_{ij} meaning $x_i - x_j \leq c_{ij}$ for clocks $x_i, x_j \in X \cup \{\mathbf{0}\}$. Extending the data structure to a priced DBM, we add an affine hyperplane. The offset point is the unique valuation such that all valuations in the zone are component-wise equal or larger than the offset point. Alternatively, the cost at the origin could be given.

Definition 6 (Priced Zone). *A priced zone over a set of clocks X is a pair (Z, f), where Z is a zone, i.e., a conjunction of constraints on clocks or differences between clocks, and f is an affine function over X providing the cost of the clock valuations satisfying the constraints of Z.*

Without going into details on how to compute the successors of a priced symbolic state, we notice that the representation of priced zones as priced DBMs support the necessary operations to do so. In particular, the data structure supports computing the set of delay successors of a priced zone and computing the projection of a priced zone (for resetting clocks). The crucial addition compared to regular DBMs is the efficient manipulation of the hyperplane in such a manner, that any state in the resulting zone is associated with the lowest cost of immediately reaching that state from a state in the predecessor. Also, given two symbolic states S and S', computing whether one dominates the other is efficiently computable. E.g. S is dominated by S' if for all states s in S, s is in S' and the cost of s in S' is lower than the cost in S.

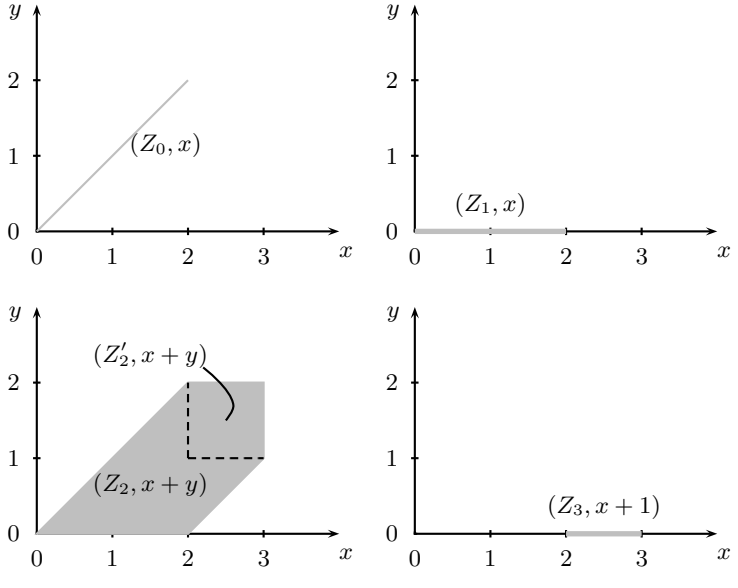


Fig. 4. Symbolic exploration of A using priced zones

Example 3. Figure 4 illustrates a symbolic exploration of the priced timed automaton A from Fig. 3 in terms of the following symbolic trace:

$$\begin{aligned}\Gamma = (l_0, (Z_0, x)) &\rightarrow (l_1, (Z_1, x)) \\ &\rightarrow (l_1, (Z_2, x+y)) \rightarrow (l_2, (Z_3, x+1))\end{aligned}$$

where $Z_0 = (x = y \wedge x \leq 2)$, $Z_1 = (y = 0 \wedge x \leq 2)$, $Z_2 = (y \leq 2 \wedge x \leq 3 \wedge 0 \leq x - y \leq 2)$ and $Z_3 = (y = 0 \wedge 2 \leq x \leq 3)$. The zone $Z_2' = (1 \leq y \leq 2 \wedge 2 \leq x \leq 3)$ is the subset of Z_2 for which the edge from l_1 to l_2 is enabled. Now, from the final symbolic state $(l_2, (Z_3, x+1))$ we see that we may reach l_2 with cost 3 given as the minimum value of the affine function $x + 1$ with respect to the constraints of the zone Z_3. This minimum value is clearly obtained at the state $(l_2, x = 2, y = 0)$. Now, we may follow this state backwards within the given symbolic trace Γ constantly selecting the predecessor-state with minimum cost. In this way we are to (re)produce the concrete minimum-cost trace:

$$\begin{aligned}\alpha_2 = (l_0, x = 0, y = 0) &\xrightarrow{1}_1 (l_0, x = 1, y = 1) \rightarrow_0 (l_1, x = 1, y = 0) \\ &\xrightarrow{1}_2 (l_1, x = 2, y = 1) \rightarrow_0 (l_2, x = 2, y = 0)\end{aligned}$$ □

```
COST := ∞
PASSED := ∅
WAITING := {S_0}
while WAITING ≠ ∅ do
    select S ∈ WAITING //based on branching strategy
    C ← infimum(S)
    if PASSED ⋠_dom S and C + remain(S) < COST then
        PASSED ← PASSED ∪ {S}
        if S ∈ GOAL then
            COST ← C
        else
            WAITING ← {S' | S' ∈ WAITING or S → S'}
return COST
```

Fig. 5. Branch and bound algorithm for cost optimal reachability analysis of priced timed automata. The algorithm works on priced symbolic states and uses auxiliary functions for computing the infimum cost of all states in a symbolic state and for checking whether a symbolic state dominates another symbolic state.

In UPPAAL CORA, cost-optimal reachability analysis is performed using a standard branch and bound algorithm. Branching is based on various search strategies implemented in UPPAAL CORA which, currently, are breadth-first, ordinary, random, or best depth-first with or without random restart, best-first, and user supplied heuristics. The latter enables the user to annotate locations of the model with a special variable called *heur* and the search can be ordered according to either largest or smallest *heur* value. Bounding is based on a user-supplied, lower-bound estimate of the *remaining* cost to reach the goal from each location, i.e. an admissible heuristic.

The algorithm depicted in Fig. 5 is the cost-optimal reachability algorithm used by UPPAAL CORA. It maintains a PASSED-list of symbolic states that have been explored and a WAITING-list of symbolic state that need to be explored and is instantiated with the initial symbolic state $\mathcal{S}_0$. The variable COST holds the currently best known cost of reaching the goal location; initially it is infinite. The algorithm iterates until no more symbolic states need to be explored. Inside the while-loop we select and remove a symbolic state, $\mathcal{S}$, from WAITING based on the branching strategy. If $\mathcal{S}$ is dominated by another symbolic state that has already been explored or it is not possible to reach the goal with a lower cost than COST, we skip this symbolic state. Otherwise, we add $\mathcal{S}$ to PASSED and if $\mathcal{S}$ is a goal location we update the best known cost to the best cost in $\mathcal{S}$. If not, we add all successors of $\mathcal{S}$ to WAITING and continue to the next iteration. Note that the algorithm does not terminate when the first goal location is discovered which is custom with a best-first branch and bound algorithm. The reason is that we allow various branching strategies some which do not guarantee the first found goal location to be optimal.

4 Modeling

As mentioned earlier, one of the main strengths of using priced timed automata for specifying and analyzing scheduling problems is the simplicity of the modeling aspect in terms of compositional descriptions. In this section, we show how to model well-known, generic scheduling problems, provide experimental results, and describe two industrial case studies.

Scheduling problems often consist of a set of passive objects, called resources, and a set of active objects, called tasks. The resources are passive in the sense that they provide a service that tasks can utilize. Traditionally, the scheduling problem is to complete the tasks as fast as possible using the available resources under some constraints, e.g. limited availability of the resource, no two tasks can, simultaneously, use the same resource, etc. The models we provide in this section are all cost extensions of classical scheduling problems.

A generic resource model (see Fig. 6a) is a two-location cyclic process with a single local clock, **c**. The two locations indicate whether the resource is **Idle** or **InUse**. The resource moves from **Idle** to **InUse**, when a task initiates a synchronization over the channel **start** and in the process, **c** is reset. The resource will maintain **InUse** until the clock reaches some usage time, **busy**, it then initiates synchronization over the channel **done**.

A generic task model (see Fig. 6b) is an acyclic process progressing from an initial location, **Init**, to a final location **Done**, indicating task completion. Intermediate locations describe acquiring resources and releasing them, i.e. the task will transit to state **Using** by initiating synchronization over a **start** channel and setting the **busy** variable of the resource. The task will remain here until the resource initiates synchronization using the **done** channel.

To solve the scheduling problem, we pose the reachability question of whether we can reach a state in which all tasks are in the location **Done**. In the following

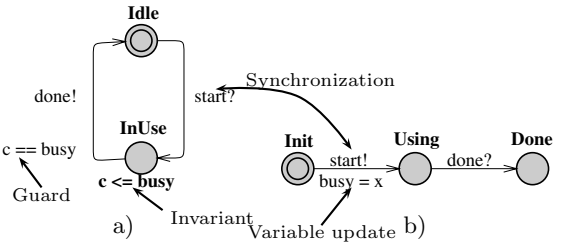


Fig. 6. a) Resource template with clock **c**. b) Task template.

four sections we present some classical scheduling problems, all of which are slight modifications of the generic templates.

4.1 Job Shop Scheduling

Problem: We are given a number of machines (resources) and a number jobs (tasks) with corresponding recipes. A recipe for a job dictates the subset of machines that the job should be processed by, the order in which the processing should happen, and the duration of each processing step. Now, the scheduling problem is to assign to each job a starting time for every required machine such that no machine is occupied by two jobs at the same time.

Cost: The model can be extended with costs by assigning to each machine an idling cost and a operation cost.

Modeling: Figure 7a depicts a job and a machine. The model of the machine is identical to the resource template, except that both locations have been extended with cost rates. The job model is a "serial composition" of the task template, i.e. the job serially requests the machines described by the recipe, in this case machines 0, 1, and 2 for 7, 5, and 15 time units, respectively.

4.2 Task Graph Scheduling

Problem: This problem is described in Section 1.

Cost: We assign to each processor an energy consumption rate while idle and while executing. Now, the overall objective is to find the schedule that minimizes the total cost while respecting a global (or task individual) deadline.

Modeling: The models for a task and a processor are depicted in Fig. 2. Again, the processor model is an exact instance of the resource template with added cost rates. Tasks 1 and 2 are exact instances of the task template, while task 3 is not. The reason is that tasks 1 and 2 can only execute on one processor each, while task 3 can execute on both, thus, task 3 is an extension of the task template with a nondeterministic choice between the processors. Furthermore, the edges leaving the initial state have been extended with a guard specifying the dependencies of the task graph, i.e. task 3 requires tasks 1 and 2 to be finished, i.e. **f**[**1**] && **f**[**2**].

4.3 Vehicle Routing with Time Windows

Problem: We are given a depot owning a fleet of vehicles (resources) with limited capacity of some good and a number of dispersed customers (tasks) with individual demands and time windows. The scheduling problem is to assign routes to each vehicle such that customers are served within their time windows and the total demand of a route does not exceed the capacity of the vehicle. Usually, the unloading process is also associated with a delay linear in the amount to unload and each vehicle is expected to return to the depot.

Cost: For a schedule, costs are incurred while vehicles are in operation (driver salary, **ds**), i.e. away from the depot, and extra costs are added while driving corresponding to the fuel consumption, **fc**.

Modeling: Figure 7b depicts models for a customer and for a vehicle. The customer model (having time window [30,90]) is a combination of the job model and the model of task 3 above. The customer can acquire either vehicle, hence the nondeterministic choice and the sequential part corresponds to acquiring the vehicle to arrive (**ComingHere**) and to unload the goods (**Unloading**). Note that besides updating the vehicle busy time with a driving distance, the vehicle capacity, **carcap**, is updated to reflect the demand. The requirements for the time window are realized through guards and invariants on the global time. The vehicle model is a slight variation of the resource template, as the **InUse** location has been replaced by two locations to distinguish between **Driving** and **Unloading**. Furthermore, there is an acyclic part reflecting the possibility of **DrivingHome** to the depot and thus completing the route. In all locations except **Home**, there is a cost rate corresponding to the driver salary and in the driving locations there is an added fuel cost.

4.4 Aircraft Landing

Problem: Given a number of aircrafts (tasks) with designated type and landing time window, assign a landing time and runway (resource) to each aircraft such that the aircraft lands within the designated time window while respecting a minimum wake turbulence separation delay between aircrafts of various types landing on the same runway.

Cost: The cost extended problem associates with each aircraft an additional target landing time corresponding to approaching the runway at cruise speed. Now, if an aircraft is assigned a landing time earlier than the target landing time, a cost per time unit is incurred, corresponding to powering up the engines. Similarly, if an aircraft is assigned a later landing time than the target landing time a cost per time unit is added corresponding to increased fuel consumption while circling above the airport.

Modeling: Figure 7c depicts a runway that can handle aircrafts of types B747 and A420, and an aircraft with target landing time 153, type A420 and time window [129,559]. Unlike the other models, the runway model has only a single location in its cycle indicating both that the resource is **IdleAndInUse**. A single location is used since the duration that a runway is occupied

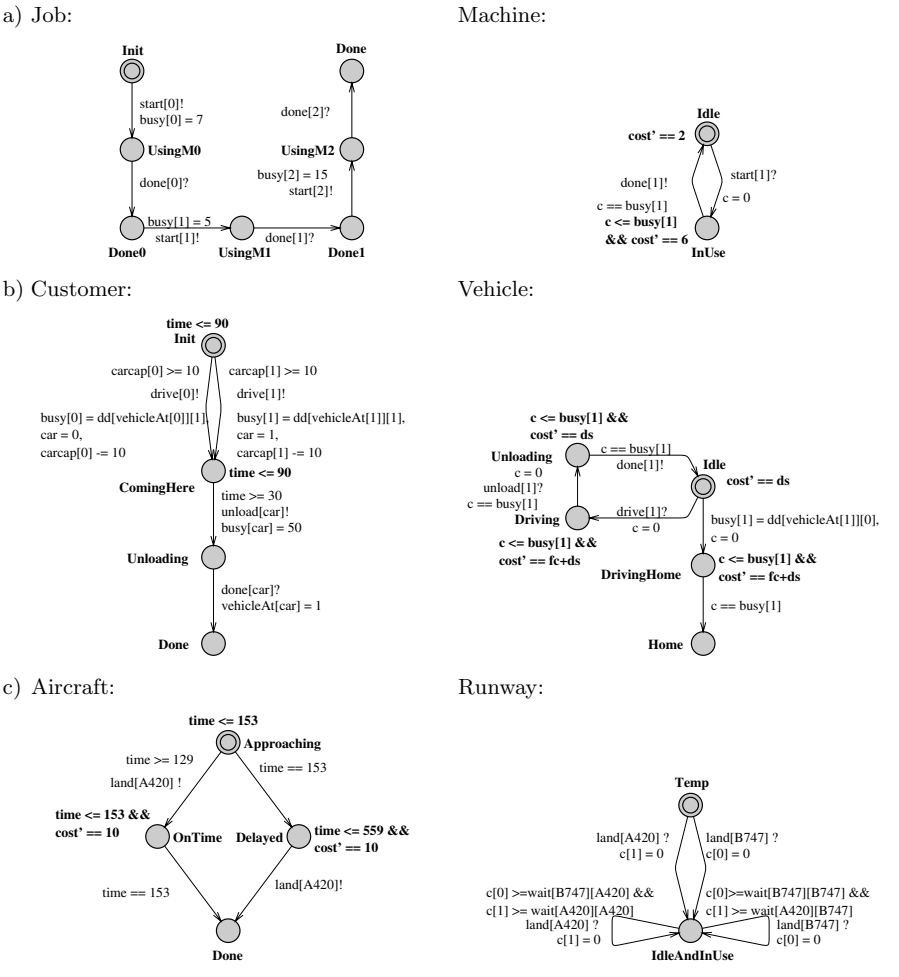


Fig. 7. Priced timed automata models for two classical scheduling problems

depends solely on the types of consecutively landing aircrafts. Thus, the runway maintains a clock per aircraft type holding the time since the latest landing of an aircraft of the given type and access to the runway is controlled by guards on the edges. The nondeterminism of the aircraft model does not distinguish between the runway to use, but whether to land early ([129,153]) or late ([153,559]). Choosing to land early, the aircraft model moves to the **OnTime** location and must remain here until the target landing time while incurring a cost rate per time unit for landing early, similarly, the aircraft can choose to land late and move to **Delayed** may remain there until the latest landing time while paying a cost rate for landing late.

4.5 PTA Versus MILP

We only provide experimental results for the aircraft landing problem comparing the PTA approach to that of MILP. For performance results of the job shop and task graph scheduling problems, we refer to [6,23,1].

RW	Planes	10	15	20	20	20	30	44
	Types	2	2	2	2	2	4	2
1	MILP (s)	*0.4*	*5.2*	*2.7*	*220.4*	*922.0*	*33.1*	*10.6*
	MC (s)	**0.8**	**5.6**	**2.8**	**20.9**	**49.9**	**0.6**	**2.2**
	Factor	*2.0*	*1.08*	*1.04*	**10.5**	**18.5**	**55.2**	**48.1**
2	MILP (s)	*0.6*	*1.8*	*3.8*	*1919.9*	*11510.4*	*1568.1*	*0.2*
	MC (s)	**2.7**	**9.6**	**3.9**	**138.5**	**187.1**	**6.0**	**0.9**
	Factor	*4.5*	*5.3*	*1.02*	**13.9**	**61.5**	**261.3**	*4.5*
3	MILP (s)	*0.1*	*0.1*	*0.2*	*2299.2*	*1655.3*	*0.2*	N/A
	MC (s)	**0.2**	**0.3**	**0.7**	**1765.6**	**1294.9**	**0.6**	
	Factor	*2.0*	*3.0*	*3.5*	**1.30**	**1.28**	*3.0*	
4	MILP (s)	N/A	N/A	N/A	*0.2*	*0.2*	N/A	N/A
	MC (s)				**3.3**	**0.7**		
	Factor				*16.5*	*3.5*		

Fig. 8. Computational result for the aircraft landing problem using PTA and MILP on comparable machines

Figure 8 displays experimental results for various instances of the aircraft landing problem using MILP and PTA. The results for MILP have been taken from [4] and the results for PTA have been executed on a comparable computer. Factors in bold indicate the performance difference in favor of PTA and similarly for italics and MILP. The experiments clearly indicate that PTA is a competitive approach to solving scheduling problems and for one non-trivial instance it is even more than a factor 250 faster than the MILP approach. However, the required computation time of the PTA approach grows exponentially with the number of added runways (and thus clocks) while no similar statement can be made for the MILP approach. The exponential growth of the PTA approach is no surprise as reachability is exponential in the number of clocks. However, this does not mean that PTA are unsuited for larger problems, but merely that the models should be carefully considered to minimize the number of clocks. Furthermore, techniques from timed automata theory to deal with clocks such as omitting certain "inactive" clocks from locations has been extended to PTA.

In conclusion, PTA is a promising method for solving scheduling problems, but further experiments need to be conducted before saying anything more conclusive.

4.6 Industrial Case Study: Steel Production

Problem: Proving schedulability of an industrial plant via reachability analysis of a timed automaton model was first applied to the SIDMAR steel plant,

which was included as a case study of the Esprit-LTR Project 26270 VHS (Verification of Hybrid Systems). The plant consists of five processing machines placed along two tracks and a casting machine where the finished steels leaves the system. The tracks and machines are connected via two overhead cranes. Each quantity of raw iron enters the system in a ladle and depending on the desired final steel quality undergoes treatments in the different machines for different durations. The planning problem consists in controlling the movement of the ladles of steel between the different machines, taking the topology (e.g. conveyor belts and overhang cranes) into consideration.

Performance: A schedule for three ladles was produced in [14] for a slightly simplified model using UPPAAL. In [15] schedules for up to 60 ladles were produced also using UPPAAL. However, in order to do this, additional constraints were included that reduce the size of the state-space dramatically, but also prune possibly sensible behavior. A similar reduced model was used by Stobbe [24] using constraint programming to schedule 30 ladles. All these works only consider ladles with the same quality of steel. In [6], using a search order based on priorities, a schedule for ten ladles with varying qualities of steels is computed within 60 seconds CPU-time on a Pentium II 300MHz. The initial solution found is improved by 5% within the time limit. Allowing the search to go on for longer, models with more ladles can be handled.

4.7 Industrial Case Study: Lacquer Production

Problem: The problem was provided by an industrial partner of the European AMETIST project as a variation on job shop scheduling. The task is to schedule lacquer production. Lacquer is produced according to a recipe involving the use of various resources, possibly concurrently, see Fig. 9. An *order* consists of a recipe, a quantity, an earliest starting date and a delivery date. The problem is then to assign resources to the order such that the constraints of the recipes and of the orders are met. Additional constraints are provided by the resources, as they might require cleaning when switching from one type of lacquer to another, or might require manual labor and thus are unavailable during the night or in weekends.

Cost: The cost model is similar to that of the aircraft landing problem. Orders finished on the delivery date do not incur any costs (except regular production costs which are not modeled as these are fixed). Orders finishing late are subject to *delay costs* and orders finishing too early are subject to *storage costs*. Cleaning resources might generate additional costs.

Modeling: Resources are modeled using the resource template. Resources requiring cleaning are extended with additional information to keep track of the last type of lacquer produced on the resource. Cleaning costs are typically a fixed amount and are added to the cost when cleaning is performed. Orders are modeled similarly to tasks in the task graph scheduling problem, except that multiple resources may be acquired simultaneously. Storage and delay costs are modeled similarly to costs in the aircraft landing problem.

5 Other Optimization Problems

At present Uppaal Cora supports cost-optimal location-reachability for PTAs. However, a number of other optimization problems are planned to be included in future releases. In the following we give a brief description of these extensions with illustrating examples.

Infinite Schedules

For several planning problems the objective is to *repeat* a treatment or process *indefinitely* and to do so in a cost-optimal manner. Now let $\alpha = s_0 \xrightarrow{a_1}_{p_1} s_1 \xrightarrow{a_2}_{p_2} s_2 \cdots \xrightarrow{a_n}_{p_n} s_n \cdots$ be an infinite execution of a given PTA, let c_n (t_n) denote the accumulated cost (time) after n steps (i.e. $c_n = \sum_{i=1}^{n} p_i$). Then the limit of c_n/t_n when $n \to \infty$ describes the cost per time of α in the long run and is the cost of α. The optimization problem is to determine the (value of the) optimal such infinite execution α^*.

Example 4. Consider the priced timed automaton B of Fig. 10 being a cyclic extension of the priced timed automaton A of Fig. 3. Below we offer two infinite (cyclic) traces (* indicates the nested cycle):

$$\begin{aligned}\beta_0 = (l_0, x=0, y=0) &\xrightarrow{1}_1 (l_0, x=1, y=1) \to_0 (l_1, x=1, y=0)^* \\ &\xrightarrow{2}_4 (l_1, x=3, y=2) \to_0 (l_2, x=3, y=0) \\ &\xrightarrow{1}_3 (l_2, x=3, y=1) \to_0 (l_0, x=0, y=1) \\ &\xrightarrow{1}_1 (l_0, x=1, y=2) \to_0 (l_1, x=1, y=0)^*\end{aligned}$$

$$\begin{aligned}\beta_1 = (l_0, x=0, y=0) &\to_0 (l_1, x=0, y=0)^* \xrightarrow{2}_4 (l_1, x=2, y=2) \\ &\to_0 (l_2, x=0, y=0) \xrightarrow{2}_6 (l_2, x=2, y=2) \\ &\to_0 (l_0, x=0, y=2) \to_0 (l_1, x=0, y=0)^*\end{aligned}$$

For the two infinite traces β_0 and β_1 their cyclic nature entails that the limit of cost per time is given as the ratio of cost per time of the nested cycles. Thus we find that:

$$\begin{aligned}ratio(\beta_0) &= (4+3+1)/(2+1+1) = 2 \\ ratio(\beta_1) &= (4+6)/(2+2) = 2.5\end{aligned}$$

and hence that β_0 offers the better solution. □

In [8] the problem of identifying the optimal infinite execution (and the limit-ratio of this execution) has been shown decidable for PTA using a so-called "corner-point" abstraction which is an extension of the classical region-technique for timed automata. In case of non-strict guards — as is the case of the priced timed automaton of Fig. 10 — the "corner-point" abstraction is identical to the

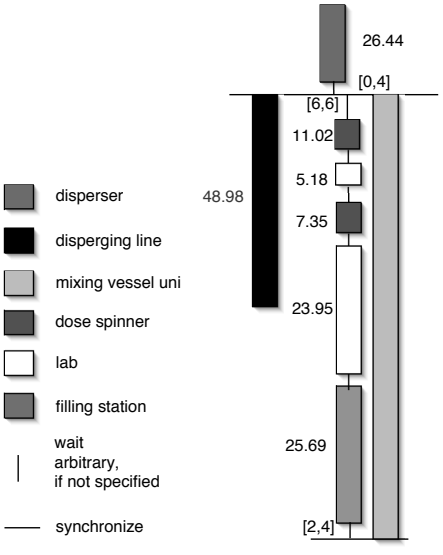


Fig. 9. A lacquer recipe. Each bar represents the use of a resource. Horizontal lines indicate synchronization points. Timing constraints for how long resources are used or separation times between the use of resources can be provided either as a fixed time or time window.

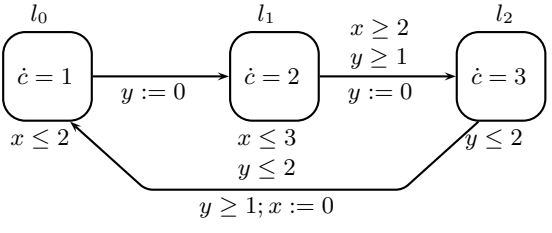


Fig. 10. A cyclic priced timed automaton, B

discrete-time semantics of the automaton. The problem now reduces to that of identifying a cycle with minimum mean-cost in the corresponding finite weighted graph, a problem for which Karp's algorithm [18] provides a cubic solution. Figure 11 illustrates the discrete semantics of the priced timed automaton B of Fig. 10.

Though the "corner-point" abstraction technique nicely demonstrates decidability of the problem (and many other decision problems for timed automata) it does not provide a practical implementation, which is still to be identified. However, a method for determining *approximate optimal* infinite schedules have been identified and applied to the synthesis of Dynamic Voltages Scaling scheduling strategies.

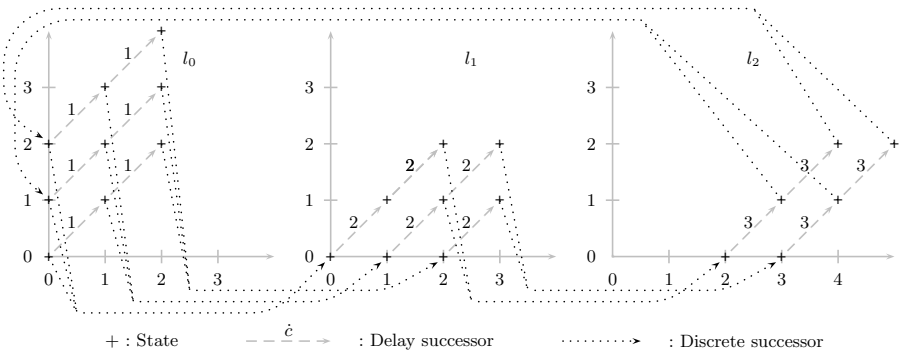


Fig. 11. Discrete time semantics for PTA, B

Multiple Cost Variables

Optimization problems may involve *multiple* cost variables (e.g. money, energy, pollution, etc.). Currently UPPAAL CORA is only capable of optimizing with respect to single costs. However, for scheduling problems with multiple costs, there might well be several optimal solutions due to "negative" dependencies between costs, i.e. minimizing one cost variable (e.g. money) might maximize others (e.g. pollution).

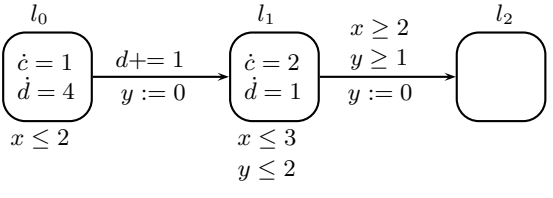


Fig. 12. A dual-priced timed automaton, C

Example 5. Figure 12 illustrates a dual-priced TA C extending the (single) priced TA A of Fig. 3 with a second cost variable d. The following two traces both reach the goal location l_2 but with incompatible cost-pairs, namely $(4, 2)$ versus $(3, 5)$.

$$\gamma_0 = (l_0, x = 0, y = 0) \rightarrow_{(0,0)} (l_1, x = 0, y = 0) \xrightarrow{2}_{(4,2)} (l_1, x = 2, y = 2) \rightarrow_{(0,0)} (l_2, x = 2, y = 0)$$

$$\gamma_1 = (l_0, x = 0, y = 0) \xrightarrow{1}_{(1,4)} (l_0, x = 1, y = 1) \rightarrow_{(0,0)} (l_1, x = 1, y = 0) \xrightarrow{1}_{(2,1)} (1_1, x = 2, y = 1) \rightarrow_{(0,0)} (l_2, x = 2, y = 0)$$ □

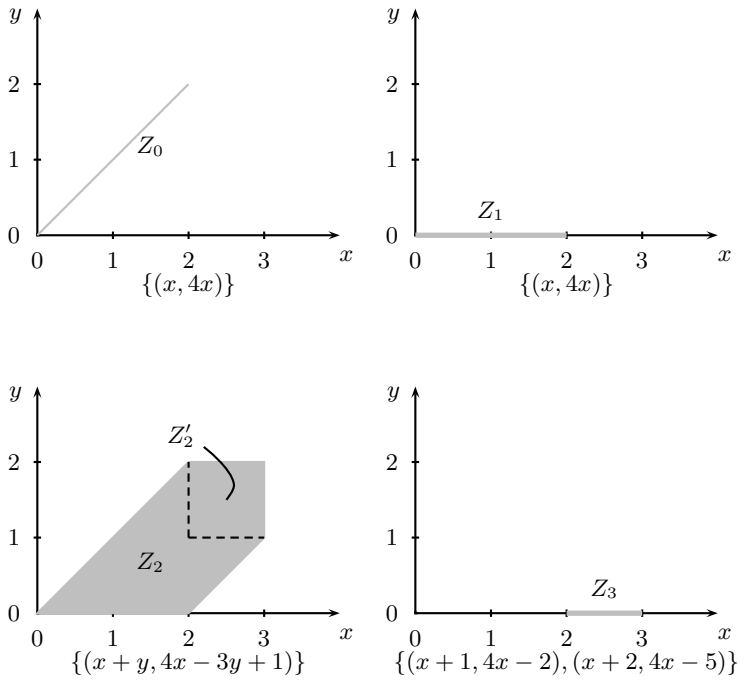


Fig. 13. Symbolic exploration of C using dual-priced zones

In [21] the notion of priced zone for PTA has been extended to multi-price TA allowing efficient synthesis of solutions optimal with respect to a chosen *primary* cost variable, but subject to user-specified upper bounds on the remaining *secondary* cost variables. More precisely, the symbolic exploration of multi-priced TAs uses multi-priced zones of the type $P = (Z, \{\mathbf{c}_1, \ldots, \mathbf{c}_n\})$, where $\mathbf{c}_i$ is a vector of affine cost-functions (one for each cost variable). Now, any concrete trace to a given state will be associated with a cost-vector giving a value (cost) for each of the involved cost variables. As illustrated by Example 5, two different traces to a given state may, in the multi-priced case, be associated with incomparable cost-vectors in the sense that neither one dominates the other with respect to component-wise $\leq$. In our symbolic treatment we deal with this phenomenon by associating the zone Z with *sets* of cost-function vectors. Now for any clock-valuation $u \in Z$ the multi-priced zone P will associate not only the set of cost-vectors $\{\mathbf{c}_1(u), \ldots, \mathbf{c}_n(u)\}$ but also all convex combinations of these vectors. We refer the interested reader to [21] for more information on this. In the following example we try to illustrate our symbolic treatment.

Example 6. Figure 13 illustrates the symbolic exploration of the dual-priced TA C of Fig. 12. In the final symbolic state the zone Z_3 is associated with a set containing two cost-function pair: $\{(x+1, 4x-2), (x+2, 4x-5)\}$. Now evaluating these two pairs with respect to the extrema points of Z_3 we obtain a set of 4

cost-pairs: $\{(3,6),(4,10),(4,3),(5,7)\}$, all combinations of which are (according to the theory) realizable. Thus, in case we want to minimize c subject to the condition that d stays below 4 it can be seen that $c = \frac{11}{4}$ is the minimum such value. □

Uncertainty

Finally, scheduling problems may involve *uncertainties* due to certain actions being under the control of an adversary. In this case the (optimal) scheduling problem is a game-theoretic problem consisting of determining a winning and optimal strategy for how to respond to any action chosen by this adversary. In [9] the problem of synthesizing optimal, winning strategies for priced timed games has been shown to be computable under certain non-zenoness assumptions. However, the problem is not solvable using zone-based technology, but needs general polyhedral support in order to represent the optimal strategies (see [10] for a methodology using HyTech).

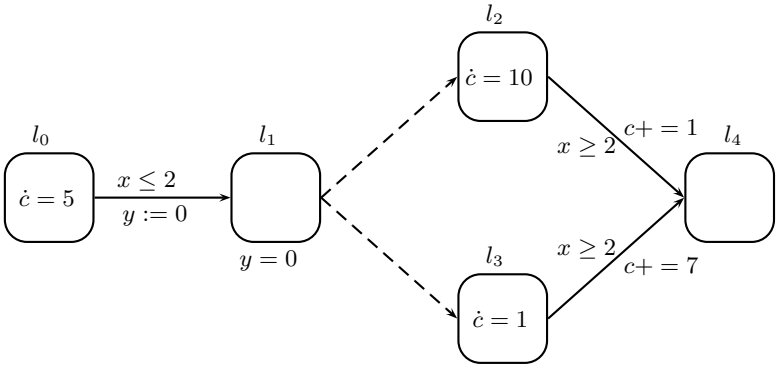


Fig. 14. A priced timed game automaton, D

Example 7. Consider the priced timed game automaton D of Fig. 14. Here the cost-rates in locations l_0, l_2 and l_3 are $5, 10$ and 1 respectively. In l_1 the adversary may choose to move to either l_2 or l_3 (dashed arrows are under control of the adversary). However, due to the invariant $y = 0$ this choice must be made instantaneously. Obviously, once l_2 or l_3 has been reached the optimal strategy for the controller is to move to the goal location l_4 immediately. Note that there is a discrete cost (respectively 1 and 7) on each discrete transition. The crucial (and only remaining) question is how long the controller should wait in l_0 before taking the transition to l_1. Obviously, in order for the controller to win this duration must be no more than two time units. However, what is the optimal choice for the duration in the sense that the overall cost of reaching l_4 will be minimal? Denote by t the chosen delay in l_0. Then $5t + 10(2-t) + 1$ is the minimal cost through l_2 and $5t + (2-t) + 7$ is the minimal cost through l_3.

As the adversary chooses between these two transitions the best choice for the controller is to delay $t \leq 2$ such that $max(21 - 5t, 9 + 4t)$ is minimum, which is obtained for $t = \frac{4}{3}$ giving a minimal cost of $14\frac{1}{3}$. □

References

1. Y. Abdeddaim, A. Kerbaa, and O. Maler. Task graph scheduling using timed automata. *In Proc. of International Parallel and Distributed Processing Symposium*, pages 8–15, 2003.
2. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
3. R. Alur, S. La Torre, and G. Pappas. Optimal paths in weighted timed automata. *Lecture Notes in Computer Science*, 2034:pp. 49–62, 2001.
4. J. E. Beasley, M. Krishnamoorthy, Y. M. Sharaiha, and D. Abramson. Scheduling aircraft landings - the static case. *Transportation Science*, 34(2):pp. 180–197, 2000.
5. G. Behrmann, A. David, and K. Larsen. A tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems*, number 3185 in Lecture Notes in Computer Science, pages 200–236. Springer Verlag, 2004.
6. G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, and J. Romijn. Efficient guiding towards cost-optimality in Uppaal. In *Proc. of Tools and Algorithms for the Construction and Analysis of System.s*, number 2031 in Lecture Notes in Computer Science, pages 174–188. Springer–Verlag, 2001.
7. Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim G. Larsen, Paul Pettersson, Judi Romijn, and Frits Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. In *Proc. of Hybrid Systems: Computation and Control*, number 2034 in Lecture Notes in Computer Sciences, pages 147–161. Springer–Verlag, 2001.
8. P. Bouyer, E. Brinksma, and K. Larsen. Staying alive as cheaply as possible. In *Proc. of Hybrid Systems: Computation and Control*, volume 2993 of *Lecture Notes in Computer Science*, pages 203–218. Springer–Verlag, 2004.
9. P. Bouyer, F. Cassez, E. Fleury, and K. Larsen. Optimal strategies in priced timed game automata. In *Proc. of Foundations of Software Technology and Theoretical Computer Science*, volume 3328 of *Lecture Notes in Computer Science*, pages 148–160. Springer–Verlag, 2004.
10. Patricia Bouyer, Franck Cassez, Emmanuel Fleury, and Kim G. Larsen. Synthesis of optimal strategies using hytech. In *Workshop on Games in Design and Verification*, volume 119(1) of *Electronic Notes in Theoretical Computer Science*, pages 11–31, Boston, MA, USA, July 2004. Elsevier Science Publishers.
11. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In *Proc. of Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer-Verlag, 1998.
12. UPPAAL CORA. `http://www.cs.aau.dk/~behrmann/cora`, Jan. 2005.
13. D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Proc. Of Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notex in Computer Science*, pages 197–212. Springer–Verlag, 1989.
14. A. Fehnker. Scheduling a steel plant with timed automata. In *Proc. of Real-Time and Embedded Computing Systems and Applications.*, page 280. IEEE Computer Society, 1999.

15. T. Hune, K. Larsen, and P. Pettersson. Guided synthesis of control programs using Uppaal. *Nordic Journal of Computing*, 8(1):43–64, 2001.
16. IF. `http://www-verimag.imag.fr/~async/IF`, Jan. 2005.
17. Advanced Methods in Timed Systems (AMETIST). `http://ametist.cs.utwente.nl`, Jan. 2005.
18. R. M. Karp. A characterization of the minimum mean-cycle in a digraph. *Discrete Mathematics*, 23(3):309–311, 1978.
19. K. Larsen, G. Behrmann, E. Brinksma, A. Fehnker, T. Hune, P. Pettersson, and J. Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In *Proc. of Computer Aided Verification*, volume 2102, pages pp. 493+, 2001.
20. K. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.
21. K. Larsen and J. Rasmussen. Optimal conditional reachability for multi-priced timed automata. In *Proc. of Foundations of Software Science and Computation Structures*, volume 3441 of *Lecture Notes in Computer Science*, pages 234–249. Springer–Verlag, 2005.
22. Verification of Hybrid Systems (VHS). `http://www-verimag.imag.fr/VHS/`, Jan. 2005.
23. J. Rasmussen, K. Larsen, and K. Subramani. Resource-optimal scheduling using priced timed automata. In *Proc. of Tool and Algortihms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 220–235. Springer Verlag, 2004.
24. M. Stobbe. Results on scheduling the sidmar steel plant using constraint programming. Internal report, 2000.
25. UPPAAL. `http://www.uppaal.com`, Jan. 2005.

rCOS: Refinement of Component and Object Systems*

Zhiming Liu[1], He Jifeng[1,**], and Xiaoshan Li[2]

[1] International Institute for Software Technology,
United Nations University, Macao SAR, China
{lzm, hjf}@iist.unu.edu
[2] Faculty of Science and Technology,
University of Macau, Macao SAR, China
xsl@umac.mo

Abstract. We present a model of object-oriented and component-based refinement. For object-orientation, the model is *class-based* and refinement is about *correct* changes in the structure, methods of classes and the main program, rather than changes in the behaviour of individual objects. This allows us to prove refinement laws for both high level design patterns and low level refactoring. For component-based development, we focus on the separation of concerns of *interface* and *functional contracts*, leaving refinement of *interaction protocols* in future work. The model supports the specification of these aspects at different levels of abstractions and their consistency.

Based on the semantics, we also provide a general definitional approach to defining different relational semantic models with different features and constraints.

Keywords: Object-Orientation, Component-Based Development, Refinement, Specification, Consistency.

1 Introduction

Today's software engineering is mainly concerned with systematic development of large and complex systems. To cope with the scale of the problem traditional software engineers divide the problem along three axes:

1. Along the temporal axis the development activities are divided into three stages of requirements specification, design and implementation.
2. Different activities in each stage deal with different aspects of the system. Requirement analysis is split into specification of aspects of static data structure, control flows or processes and operations or services. Similarly, design may be split into design strategies for concurrency, design strategies for efficiency and design strategies for security. These strategies are commonly expressed as design patterns [13]. Finally implementation may be split into databases, user interfaces and libraries for security.

* This is a revised and extended version of the combination of the papers [17,31]. This work is partly supported by e-Macao project funded by the Government of Macao, and the research grant 02104 MoE and the 973 project 2002CB312000 of MoST of P.R. China.

** On leave from East China Normal University, Shanghai, China.

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 183–221, 2005.
© Springer-Verlag Berlin Heidelberg 2005

3. The third axis is that of system evolution and maintenance [20,24] where each evolutionary or maintenance step enhances the system by iterating through the requirements to implementation cycle.

Unfortunately in practical software engineering, all aspects are specified using informal techniques and therefore this approach does not give the desired assurances and productivity. The main problems are, among others, the following:

- Since the requirements specification is informal there is no way to ascertain its completeness resulting in a lot of gaps.
- The gaps in requirements are filled by ad-hoc decisions taken by programmers who are not qualified for the job of requirement analysis. This results in code of poor quality.
- There is no traceability between requirements and implementation making it very expensive to accommodate changes and maintain the system.
- Most of the tools are for project management and system testing. Although these are useful, they are not enough for ensuring the semantic correctness of the implementation for a requirements specification and semantic consistency of changes made in the system.

Formal methods, on the other hand, attempt to complement the informal engineering methods by techniques for formal modelling, specification, verification and refinement [46,15]. Formal methods were born and has grown up in those years of structured analysis and design. So we have theories of formal specification, verification, refinement, decomposition and composition. In principle, a formal system development starts with an abstract specification and transforms it into a program through a number of refinement steps. A formal method is supported by a sound logical framework. These have helped in improving the quality of software systems so that they are more correct and safer to use. However, they do not yet support the three dimensional development process well. It is still a great challenge to scale up formal methods to industry scale because of the problems listed below.

- Formal methods have inherited the same disadvantages from the one-dimensional "water-fall model" of development activities. They suffer even more seriously from those disadvantages as a specification of the whole system at any level, e.g. the requirement level, in a formal notation is not understandable to most system engineers, not to mention about formal verification.
- Because of the theoretical goal of completeness and independence, refinement calculi, including those for object-oriented programming [22,23,1,8], provide only refinement rules for a small change in each step. Refinement calculi can therefore be difficult to be used in practice. Data refinement always requires definition of a semantic relation between the programs (their state space) and is hard to be applied systematically and to deal with "big-step refinement".
- There is no clear separation of concerns making it difficult for domain experts, architects and programmers to collaborate towards a single solution. The existing object-oriented models, e.g. [22,23,1,8], focus on much programming aspects and it is not clear what kind of properties of an object-oriented program can be described

and proven in such a model. Therefore, these models cannot be used for software development as we do not know from what a specification that the system is to be developed or refined.
- It is not easy for software engineers to build correct and proper models, from low level designs or implementations, that can be verified by model checking tools.
- There is no explicit support for productivity enhancing techniques such as component based development.

So far, both the formal methods and the methods adopted by practical software engineers are far from meeting the quality and productivity needs of the industry. The industry continues to be plagued by high development and maintenance costs and poor quality. However, recently there have been encouraging developments in both approaches. The software engineering community has started using precise models for early requirements analysis and design [37,12]. Theories and methods for object-oriented, component-based and aspect-oriented modelling and development are gaining attention of the formal methods community[1]. There are various attempts at investigating formal aspects of object-oriented refinement, design patterns, refactoring and coordination [5,17,31].

In this article, we present a calculus of **R**efinement of **C**omponent and **O**bject **S**ystems (rCOS)[2]. It captures the essential features of object-orientation including *reference types, inheritance, dynamic binding, and visibility*. Unlike the object logic in [1], rCOS is *class-based* and refinement is about *correct* changes in the structure, methods of classes and the main program, rather than changes in the behaviour of individual objects. For component-based development, we are concerned with the separation of concerns of *interfaces*, *functional contracts* and *interaction protocols*. In this paper, we focus on *interfaces* and *functional contracts*. rCOS reflects the basic object-oriented and component-based principles of *component and class decomposition*, *task delegation* and *data encapsulation*. It allows refinement for both high level *design patterns* and low level *refactoring* [34].

We briefly introduce in Section 2, as our semantic basis, the notion of *designs* in Unifying Theories of Programming [19]. In Section 3, we define the model for object systems. We present refinement calculus of *object-oriented designs* in Section 4.1. We then show in Section 5 how the model for object systems is extended to deal with component systems. In Section 6, we conclude the article with discussion and related work.

2 Semantic Basis

We take a classical approach to modelling the execution of a program in terms of a relation between the *states* of the program. However, the concept of state is more general than what programmers usually understand and it depends on what the modeler

[1] In addition to the well-established conferences such OOPSLA and ECOOP, many new conferences and workshops on component systems, object systems and aspect-oriented techniques.

[2] rCOS is produced by LaTeX command {\large r}\textsc{COS}. In [17], the calculus was named as OOL.

wants to observe of the execution of a program. For example, for a terminating sequential program, we are only interested in the initial inputs and final outputs. For a possible non-terminating program, we need an observable by which we can describe if the program terminates for some inputs. For concurrent and communicating program, we would like to observe the possible *traces* of interactions, *divergencies* and *refusals*, in order to verify if program is deadlock free and livelock free. If we are interested in real-time programs, we need to observe the time. Identification of what to observe in different kinds of systems is one of the core ideas of the Unifying Theories of Programming [19].

We call what to be observed of a program *P* the *observables* or *alphabet* of the program, denoted by $\alpha(P)$ and simply α when there is no confusion. An observable of *P* may take different values for different executions or runs, but from the same value space called the *type* of the observable. Therefore, an observable is also a *variable*. Though not all observables have to appear in a program text, but they are all needed to define the semantics of the program.

Given an alphabet α, a *state* of α is a (well-typed) mapping from α to the value spaces of the observables. A program *P* with an alphabet α is then defined as a pair of predicates, called a *design* and represented as $Pre \vdash Post$, with free variables in α. It means that if the value of observables satisfies the *precondition Pre* at the beginning of the execution, the execution will *generate* observables satisfying the *postcondition Post*, and thus defined as the implication

$$(Pre \vdash Post) \stackrel{def}{=} Pre \Rightarrow Post$$

2.1 Programs as Designs

This subsection briefly shows how the basic programming constructs can be defined as designs. For details, we refer the reader to [19].

For an imperative sequential program, we are interested in observing the values of the input variables $in\alpha$ and output variables $out\alpha$. Here we take the convention that for each input variable $x \in in\alpha$, its primed version x' is in an output variable in $out\alpha$, that gives the final value of x after the execution of the program. We use a Boolean variable *ok* to denote whether a program is *started properly* and its primed version ok' to represent whether the execution has terminated. The alphabet α is defined as the union $in\alpha \cup out\alpha \cup \{ok, ok'\}$, and a design is of the form

$$(p(x) \vdash R(x, x')) \stackrel{def}{=} ok \wedge p(x) \Rightarrow ok' \wedge R(x, x')$$

where

- p is a predicate over $in\alpha$ and R is a predicate over $out\alpha$,
- p is the *precondition*, defining the initial states
- R is the *postcondition*, relating the initial states to the final states.
- ok and ok': describe the termination, they do **not** appear in expressions or assignments of program texts

The design represents a *contract* between the "user" and the program such that if the program is started properly in a state satisfying the precondition it will terminate in a state satisfying the postcondition R.

A design is often *framed* in the form

$$\beta : (p \vdash R) \stackrel{def}{=} p \vdash (R \wedge \underline{w}' = \underline{w})$$

where $\underline{w}$ contains all the variables in $in\alpha$ but those in β.

Before we define the semantics of a program, we first define some operations on designs.

- Given two designs such that the output alphabet of P is the same as primed version of the input alphabet of Q, the sequential composition

$$P(in\alpha_1, out\alpha_1); Q(in\alpha_2, out\alpha_2) \stackrel{def}{=} \exists m \cdot P(in\alpha_1, m) \wedge Q(m, out\alpha_2)$$

- Conditional choice: $(D_1 \lhd b \rhd D_2) \stackrel{def}{=} (b \wedge D_1) \vee (\neg b \wedge D_2)$
- Demonic and angelic choice operators:

$$D_1 \sqcap D_2 \stackrel{def}{=} D_1 \vee D_2 \quad D_1 \sqcup D_2 \stackrel{def}{=} D_1 \wedge D_2$$

- `while` b `do` D is defined as the weakest fixed point

$$X = ((D; X) \lhd b \rhd \mathit{skip})$$

We can now define the meaning of primitive commands program commands as framed designs in Table 1. Composite statements are then defined by the operations on designs.

Table 1. Basic commands as designs

command: *c*	**design: $[\![c]\!]$**	**description**
skip	$\{\} : \mathit{true} \vdash \mathit{true}$	does not change anything, but terminates
chaos	$\{\} : \mathit{false} \vdash \mathit{true}$	any thing, including non-terminating, can happen
$x := e$	$\{x\} : true \vdash x' = val(e)$	side-effect free assignment; updates x with the value of e
$m(e; v)$	$[\![$`var` *in*, *out*$]\!]$; $[\![$*in*:=*e*$]\!]$; $[\![body(m)]\!]$; $[\![$*v*:=*out*$]\!]$; $[\![$`end` in, $out$$]\!]$	$m(in; out)$ is the signature with input parameters *in* and output parameters *out*; $body(m)$ is the body command of the procedure/method

In general, when defining a particular programming language, the preconditions are usually strengthen with some *well-definedness* conditions of the commands, and a program or command c is generally of the form

$$[\![c]\!] \stackrel{def}{=} D(c) \Rightarrow \mathit{Spec}$$

where $Spec$ is a design. Some of the well definedness conditions may even be dynamic.

Strengthening precondition with well-definedness conditions allows us to treat correcting a unwell-defined command to a well-formed one as refinement. This is essential to support incremental and iterative development as most cases of unwell-defined are due to the insufficiency of data or services. Therefore, adding more data, services and components, without altering the existing ones, will be refinement in our framework.

In this article, we will add variables about dynamic typing, visibility, etc, to define object-oriented programs. This ensures that the logic of rCOS is a conservative extension to that for imperative programs. Therefore, all the laws about imperative commands will remain valid without the need of reproving. rCOS can be further extended to deal with features of communication, interaction, real-time and resources. If we adding variables for traces, refusals and divergencies into the alphabet, different kinds of semantics of communicating processes can be defined as designs [19,10]. Also, using clock variables in the alphabet, we can define real-time programs as designs too [43]. It is possible to further extend rCOS to describe resource consumptions, such as memory and processor nodes, by introducing resource variables [21].

2.2 Refinement of Designs

The refinement relation between designs is then defined to be logical implication. A design $D_2 = (\alpha, P_2)$ is a **refinement** of design $D_1 = (\alpha, P_1)$, denoted by $D_1 \sqsubseteq D_2$, if P_2 entails P_1 if

$$\forall x, x', \ldots, z, z' \cdot (P_2 \Rightarrow P_1)$$

where $x, x', \ldots, z, z'$ are variables contained in α. We write $D_1 = D_2$ if they refine each other.

If they do not have the same alphabet, we can use data refinement. Let ρ be a mapping from α_2 to α_1. Design $D_2 = (\alpha_2, P_2)$ is a **refinement** of design $D_1 = (\alpha_1, P_1)$ under ρ, denoted by $D_1 \sqsubseteq_\rho D_2$, if $(\rho; P_1) \sqsubseteq (P_2; \rho)$. It is easy to prove that *chaos* is the worst program, i.e. $chaos \sqsubseteq P$ for any program P. For more algebraic laws of imperative programs, please see [19].

The following theorem is the basis for the fact that the notion of designs can be used for defining a semantics of programs.

Theorem 1. *The notion of designs is closed under programming constructors:*

$$((p_1 \vdash R_1); (p_2 \vdash R_2)) = ((p_1 \wedge \neg(R_1; \neg p_2)) \vdash (R_1; R_2))$$
$$(p_1 \vdash R_1) \sqcap (p_2 \vdash R_2) = (p_1 \wedge p_2) \vdash (R_1 \vee R_2)$$
$$(p_1 \vdash R_1) \sqcup (p_2 \vdash R_2) = (p_1 \vee p_2) \vdash ((p_1 \Rightarrow R_1) \wedge (p_2 \Rightarrow R_2))$$
$$((p_1 \vdash R_1) \lhd b \rhd (p_2 \vdash R_2)) = ((p_1 \lhd b \rhd p_2)) \vdash (R_1 \lhd b \rhd R_2)$$

3 Object Systems

In this section we introduce to the syntax and semantics of rCOS for object systems.

3.1 Syntax

In rCOS, an object system (or program) *S* is of the form *Cdecls* • *Main*, consisting of class declaration section *Cdecls* and a main method *Main*. The main method is a pair (glb, c) of a finite set *glb* of *global variables declaration* and a command c. The class

declaration section *Cdecls* is a finite sequence of class declarations $cdecl_1; \ldots; cdecl_k$, where each class declaration $cdecl_i$ is of the following form

$$
\begin{array}{l}
[\textbf{private}]\ \textbf{class}\ M\ [\textbf{extends}\ N]\ \{ \\
\quad \texttt{private}\quad U_1\ a_1 = u_1, \ldots, U_m\ a_m = u_m; \\
\quad \texttt{protected}\ V_1\ b_1 = v_1, \ldots, V_n\ b_n = v_n; \\
\quad \texttt{public}\quad W_1\ d_1 = w_1; \ldots W_k\ d_k = w_k; \\
\quad \texttt{method}\quad m_1(\underline{T}_{11}\ \underline{x}_1; \underline{T}_{12}\ \underline{y}_1; \underline{T}_{13}\ \underline{z}_1)\{c_1\}; \\
\qquad\qquad \cdots; \\
\qquad\qquad m_\ell(\underline{T}_{\ell 1}\ \underline{x}_\ell; \underline{T}_{\ell 2}\ \underline{y}_\ell; \underline{T}_{\ell 3}\ \underline{z}_\ell)\{c_\ell\} \\
\quad \}
\end{array}
$$

Note that

- Each part in the body of the declaration is optional too.
- A class can be declared as *private* or *public*, but by default it is assumed to be *public*. We can understand the class section as a *Java-like package* and *Main* as an application program using the package. Only a public class or a primitive type can be used in the global variable declarations *glb* of *Main*. Later in Section 4.1, *structural refinement* laws allow us to add, delete, change (e.g. adding, deleting or changing attributes or methods), decomposing or composing private classes and associations among them without changing the behaviour of the system. Refinement is also allowed for consistent change in public classes and the main method.
- *N* and *M* are distinct names of classes, and *M* is called the direct superclass of *N*.
- Attributes annotated with *private* are private attributes of the class, and similarly, the *protected* and *public* declarations for the protected and public attributes. Types and initial values of attributes are also given in the declaration.
- The *method* declaration declares the methods, their value parameters ($\underline{T}_{i1}\ \underline{x}_i$), result parameters ($\underline{T}_{i2}\ \underline{y}_i$), value-result parameters ($\underline{T}_{i3}\ z_i$) and bodies (c_i). The body of a method c_i is a command that will be defined later.

We will use Java convention to write a class specification, and assume an attribute *protected* when it is not tagged with *private* or *public*. We have these different kinds of attributes to show how visibility issues can be dealt with. We can have different kinds of methods too for a class. However, we omit the declaration of private or public methods for the simplicity of the theory. Instead, we assume all methods are public and can be inherited by a subclass.

Symbols

To make the presentation precise we assume the following disjoint infinite sets of symbols,

- *VNAME* denotes the set of symbols of variables names and we use *x*, *y*, and *z* and their versions with subscripts when we talk about arbitrary variables.
- *CNAME* is used for the set of class names. We use *C*, *D*, M and N with possible subscripts to range over this set.
- *ANAME* is the set of symbols to be used as names of attributes, ranged over by a with possible subscripts.

Commands

rCOS supports typical object-oriented programming constructs, but it also allows some commands for the purpose of specification and refinement:

$$c ::= \mathit{skip} \mid \mathit{chaos} \mid \texttt{var}\ T\ x = e \mid \texttt{end}\ x \mid c; c \mid c \lhd b \rhd c \mid c \sqcap c \\ \mid b * c \mid le.m(\underline{e}, \underline{v}, \underline{u}) \mid le := e \mid C.new(x)$$

where b is a Boolean expression e is an expression, and le is an expression which may appear on the left hand side of an assignment and is of the form $le ::= x \mid le.a$ where x is a simple variable and a an attribute. Unlike [41] that introduces "statement expressions", we use $le.m(\underline{e}; \underline{v}; \underline{u})$ to denote a call of method m of the object denoted by the left-expression *le*. Expressions $\underline{e}$, $\underline{v}$ and $\underline{u}$ are the actual value input parameters result parameters and actual value-result parameters, respectively. They can be changed during the execution of the method call and with final output returned in the actual result and value-result parameters. The command $C.new(x)$ is to create a new object of class *C* with the initial values of its attributes as declared in *C* and assign it to variable x. Thus, $C.new(x)$ uses x with type *C* to store the newly created object.

Expressions

Expressions, which can appear on the right hand sides of assignments, are constructed according to the rules below.

$$e ::= x \mid a \mid \mathit{null} \mid e.a \mid (C)e \mid f(e)$$

where *null* represents the special object of the special class *NULL* and has *null* as its unique object, $e.a$ is the a-attribute of e, and $(C)e$ is type casting. Notice that we do not define *NULL* as a subclass of any other class as we do not allow multiple inheritance.

3.2 Semantics

rCOS adopts an observation-oriented and relational semantics. To formalize the behavior of an object-oriented program, we have to take into account the following features:

- A program operates not only on variables of primitive types, such as integers, Boolean values, but also on objects of reference types.
- To protect attributes from illegal accesses, the model has to address the problem of visibility of attributes to the environment.
- An object can be associated with any subclass of its originally declared one. To validate expressions and commands in a dynamic binding environment, the model must keep track of the current type of each object.
- The dynamic type *M* of an object can be casted up to any superclass *N* and later casted down to any class which is a subclass of *N* and a superclass of *M* or *M* itself. We therefore need to record both the casted type *N* and the dynamic type M of the object.

Static Semantics. The class declaration section *Cdecls* of a program defines the types (value space) and static structure of the program:

- *pricname*: the set $\{C \mid C \text{ is declared in } Cdecls\}$ of the private class names declared in *Cdecls*. We also use *pubcname* to record the sets of names of the public classes in declared in *Cdecls*. Let *cname* be the union of these two sets.
- *superclass*: the partial function $\{M \mapsto N \mid \text{Class } M \text{ extends } N \text{ is declared in } Cdecls\}$, recording that N is a direct superclass of M. We define the general superclass class relation $\succ$ as transitive closure of *superclass*, and $N \succeq M$ if $N \succ M$ or $N = M$.
- *pri*, *prot*, and *pub*: they associate each class name $C \in cname$ to its private attributes $pri(C)$, protected attributes $prot(C)$, and public attributes $pub(C)$, respectively:

$$
\begin{aligned}
pri(C) &\stackrel{def}{=} \{\langle a : T, d\rangle \mid T\ a = d \text{ is (declared as) a private attribute of } C\}\\
prot(C) &\stackrel{def}{=} \{\langle a : T, d\rangle \mid T\ a = d \text{ is a protected attribute of } D \succeq C \text{ for some } D \in cname\}\\
pub(C) &\stackrel{def}{=} \{\langle a : T, d\rangle \mid T\ a = d \text{ is a public attribute of } D \succeq C \text{ for some } D \in cname\}
\end{aligned}
$$

- *op*: it associates each class $C \in cname$ to its set of methods $(op)(C)$

$$
\begin{aligned}
(op)(C) \stackrel{def}{=} \{&m \mapsto (\underline{x} : \underline{T}_1, \underline{y} : \underline{T}_2, \underline{z} : \underline{T}_3, c) \mid\\
&m(\underline{x} : \underline{T}_1; \underline{y} : \underline{T}_2; \underline{z} : \underline{T}_3)\{c\} \text{ is declared as method of } C\}
\end{aligned}
$$

We define the following notations

1. The function *attr* is the union of *pri*, *prot* and *pub*; for each C, $attr(C)$ is the set of attributes declared in C itself.
2. The function *Attr* extends $attr(C)$ for each C to include the protected and public attributes that C inherited from its super classes, i.e. $Attr(C)$ contains all attributes directly accessible in methods of C.
3. The function *AAttr* extends $attr(C)$ for each C to include all attributes of C and those of it superclasses. Thus, $AAttr(C)$ determines the whole state space of an object of class C and when an object C is created, all attributes in $AAttr(C)$ need to be initialized.
4. $init(C.a)$ denotes the initial value of attribute a of C.
5. $dtype(C.a)$ denotes the *declared type* T if $\langle a : T, d\rangle \in AAttr(C)$. This is used to calculate the declared type of an *attribute expression* in C inductively:
   (a) $dtype(a) \stackrel{def}{=} dtype(C.a)$
   (b) $dtype(e.a) \stackrel{def}{=} dtype(dtype(e).a)$

We call the tuple $\langle cname, superclass, (pri, prot, pub), op\rangle$ a *program structure*, denoted by Ω_{Cdecls}. We take the whole declaration section as a command which sets up the structure:

$$[\![Cdecls]\!] \stackrel{def}{=} \{\Omega_{Cdecls}\} : true \vdash \Omega'_{Cdecls} = \langle cname, superclass, (pri, prot, pub), op\rangle$$

Definition 1. *A class declaration section Cdecls is* **well-defined**, *denoted* $\mathcal{D}(Cdecls)$, *if the following conditions hold*

1. *each class name* $M \in cname$ *and the name of its direct superclass* N *are distinct,*
2. *if* $M \in cname$ *and* $superclass(M) = N$, *then* $N \in cname$,

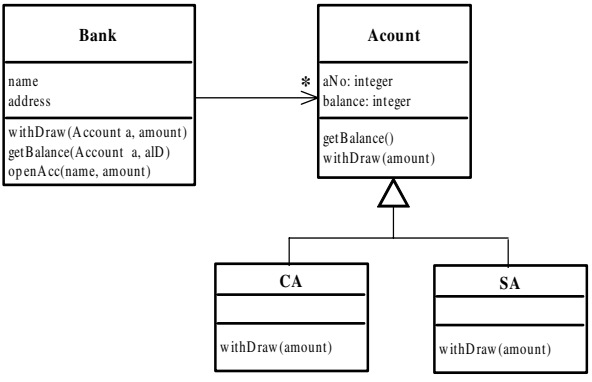


Fig. 1. A bank system

3. *any type used in declarations of attributes and parameters is either a primitive built-in type or a class in cname,*
4. *the superclass relation $\succ$ is acyclic,*
5. *any attribute of a class is not redeclared in its subclasses, i.e. we do not allow attribute hiding and alias in a subclass*[3]*,*
6. *the names of the attributes of each class are distinct,*
7. *the names of the methods of each class and the names of parameters of each methods are distinct respectively.*

A well-defined declaration section corresponds to a UML [4] class diagram, and thus it and its semantics can be used for formalisation of UML class diagrams, such as the one in Figure 1. For related work on formal support to UML-based development, we refer to our work in [32,33,47].

Type, Values and Objects. We assume a set $\mathcal{T}$ of *built-in primitive types*. We also assume an infinite set *REF* of *object identities* (or *references*), and *null* $\in$ *REF*. A *value* is either a member of a primitive type in $\mathcal{T}$ or an object identity in *REF* with its *dynamic typing information*. Let the set of values be

$$VAL \stackrel{def}{=} \bigcup \mathcal{T} \cup (REF \times CNAME)$$

For a value $v = \langle r, C \rangle \in REF \times CNAME$, we use $ref(v)$ to denote r and $type(v)$ to denote C.

Definition 2. *An* **object** o *is either the special object null, or a structure* $\langle r, C, \sigma \rangle$*, where*

- *reference* r*, denoted by* $ref(o)$*, is in REF,*
- C*, denoted by* $type(o)$*, is a class names.*
- σ *is called the* state *of* o*, denoted by* $state(o)$*, and it is a mapping that assigns each* $a \in AAttr(C)$ *to a value in* $dtype(a)$ *if* $dtype(a) \in \mathcal{T}$ *and otherwise to the null object or a value in REF* $\times$ *CNAME. We use* $o.a$ *to denote* $\sigma(a)$

[3] If we allow attribute hiding and alias, we have to introduce special object variables $this$ and $super$. We not consider this problem in this paper.

We extend the *equality* relation on values to the relation on both values and objects

$$(v_1 = v_2) \stackrel{def}{=} \begin{pmatrix} (type(v_1) = type(v_2) \wedge type(v_1) \in \mathcal{T} \wedge (v_1 = v_2)) \vee \\ \forall a \in AAttr(type(v_1)) \cdot (v_1.a = v_2.a) \end{pmatrix}$$

Notice that this equality ignores the references of objects, but only concerns about the structure and the primitive attributes of the objects in the structure.

Some Notations. Let $\mathcal{O}$ be the set of all objects, including *null*. The following notations will be employed in the semantics definitions.

- For a non-empty finite sequence of elements $\underline{s} = \langle s_1, .., s_k \rangle$, we define the head element $head(\underline{s}) = s_1$, and the tail sequence $tail(\underline{s}) = \langle s_2, .., s_k \rangle$.
- For sets S and S_1, $S_1 \rhd S$ is the set difference removing elements in S_1 from S. Let $\rhd$ have higher associativity than the normal set operators like $\cup$ and $\cap$.
- For a mapping $f : D \longrightarrow E$, $d \in D$ and $r \in E$,

$$f \oplus \{d \mapsto r\} \stackrel{def}{=} f' \qquad \text{where } f'(b) \stackrel{def}{=} \begin{cases} r, \text{ if } b = d; \\ f(b), \text{ if } b \in \{d\} \rhd D. \end{cases}$$

- For an object $o = \langle r, M, \sigma \rangle$, an attribute a of M and a value d,

$$o \oplus \{a \mapsto d\} \stackrel{def}{=} \langle r, M, \sigma \oplus \{a \mapsto d\} \rangle$$

- For a set $S \subseteq \mathcal{O}$ of objects,

$$S \uplus \{\langle r, M, \sigma \rangle\} \stackrel{def}{=} \{o \mid ref(o) = r\} \rhd S \cup \{\langle r, M, \sigma \rangle\}$$
$$ref(S) \stackrel{def}{=} \{r \mid r = ref(o), o \in S\}$$

For a given class declaration section *Cdecls*, we use Σ_{Cdecls} to denote the set of all objects of the classes declared in *Cdecls*, called the *object space* of *Cdecls*. Σ_{Cdecls} corresponds to the set of all UML *object diagrams* [4] of the UML class diagram of *Cdecls* [32]. We call the pair $(\Omega_{Cdecls}, \Sigma_{Cdecls})$ a *program context* and denote it by Ξ_{Cdecls}. When there is no confusion, we omit the subscript *Cdecls* from these notations. All the dynamic semantic definitions in the rest of this section are given under a fixed context, that is defined by a given class declaration section. Therefore the evaluation $value(e)$ of an expression e is carried out in the context Ξ and the semantics $[\![c]\!]_\Xi$ defines the state change by the execution of c in the context Ξ.

Dynamic Semantics. In rCOS, we define the behavior of an object program by a design over a set of observables or state variables. We first identify the state variables and define their states.

Variables. Now we look at what variables can be changed during the execution of the program.

System Configuration. First, we introduce a variable Π whose value is the set of objects created so far. We call Π the *current configuration* of the program in [41]. During the execution of the program, the value of Π is set in the powerset 2^{Σ} that satisfies the following conditions:

1. *objects in Π are complete*: if $o \in \Pi$ and $a \in \mathit{AAttr}(\mathit{type}(o))$ with a class type, then $o.a$ is either *null* or there is an object $o_1 \in \Pi$ and $\mathit{ref}(o.a) = \mathit{ref}(o_1)$, and
2. *Objects are uniquely identified by their references:* for any objects o_1 and o_2 in Π if $\mathit{ref}(o_1) = \mathit{ref}(o_2)$ then
   (a) $\mathit{type}(o_1) = \mathit{type}(o_2)$, and
   (b) $\mathit{ref}(\mathit{state}(o_1)) = \mathit{ref}(\mathit{state}(o_2))$, where for each $a : T \in \mathit{AAttr}(\mathit{type}(o))$
   $$\mathit{ref}(\mathit{state}(o))(a) \stackrel{def}{=} \begin{cases} \mathit{ref}(o.a) & \text{if } T \in \mathit{cname} \\ o.a & \text{if } T \in \mathcal{T} \end{cases}$$

When a new object is created or the value of an attribute of an existing object is modified, the system configuration Π will be changed. For each class C, we use variable $\Pi(C)$ to denote the set of existing objects of class C.

External Variables. A set $\mathit{glb} = \{x_1 : T_1, \ldots, x_k : T_k\}$ of variables with their types are declared in the main method of the program, where each type T_i is called the *declared type* of x_i, denoted as $\mathit{dtype}(x_i)$, and it is either a built-in primitive type or a public class in *pubcname*. Their values will be modified by methods and commands of the main method containing them.

Local Variables. A set *localvar* identifies the local variables which occur in the local variable declaration and undeclaration commands. This set includes *self* whose current value represents the current active object, parameters of methods of classes, and other variables introduced by the local declaration command. We assume that *localvar* and *glb* are disjoint.

Because method calls may be nested inside a method body, *self* and a parameter of a method may be declared a number of times with possible different types before it is undeclared. A local variable x has a sequence of declared types and is syntactically represented in the form of $(x : \langle T_1, \ldots, T_n \rangle)$. We use *TypeSeq* to denote the sequence of types of x, and T_1 is the most recently declared type of x and denoted by $\mathit{dtype}(x)$.

We use $\overline{x}$ as a variable to denote the value of a local variable x. This value comprises a finite sequence of values, whose first (head) element, which is simply denoted by x itself, represents the current value of the variable. We use the conventions that $x : \langle T \rangle$ and $\overline{x}$ for x for an external variable $x : T \in \mathit{glb}$.

Visibility. We introduce a variable *visibleattr* to hold the set of attributes which are visible to the command under execution. There the value of *visibleattr* defines the current execution environment. Before executiing a method of an object o, *visibleattr* is set to set $\mathit{Attr}(o)$ of the attributes of the current type of o, including all the declared attributes of the class, the protected and public attributes of its super classes and all public attributes of public classes; and it will be reset to the global environment consisting of all the public attributes of the public classes after the execution of the method. We will define auxiliary commands that set and reset the execution environments when we define

the semantics of a method invocation. Notice that the value space of *visibleattr* is the powerset of $\{C.a \mid C \in CNAME, a \in ANAME\}$.

We use

- *var* to denote the union of *glb* and *localvar*,
- *VAR* to denote the union of *var* plus Π and *visibleattr*, and we call it the set of *dynamic variables*,
- *glb* is the set of elements of *VAR* excluding those out of *glb*,
- for a set *V* of variables, V' to denote the set of the primed versions of the variables of *V*.

States. We now define the notion of states in the object-oriented setting.

Definition 3. *For a program $S = Cdecls \bullet Main$, a (***dynamic***) state of S is a mapping Γ from the variables VAR to their value spaces that satisfies the following conditions:*

1. *If $x \in VAR$ and $dtype(x) \in \mathcal{T}$ then $\Gamma(x)$ is a value in $dtype(x)$,*
2. *If $x \in VAR$ and $dtype(x) \in cname$ then $\Gamma(x)$ is*
   (a) *either null, or*
   (b) *a value in $v \in REF \times CNAME$ such that there exists an object $o \in \Gamma(\Pi)$ for which $ref(o) = ref(v)$ and $type(o) \preceq type(v)$.*
   This attachment of an object o to a variable x provides the information about type casting: $type(o)$ is the current *(based) type of x, denoted as $atype(x)$, and $type(v)$ is the* casted type *of x.*

Two states Γ_1 and Γ_2 are equal*, denoted by $\Gamma_1 = \Gamma_2$, if*

1. *$\Gamma_1(x) = \Gamma_2(x)$ for any $x \in VAR$ such that $dtype(x) \in \mathcal{T}$,*
2. *for any $x \in VAR$ and $dtype(x) \in cname$*
   (a) *$\Gamma_1(x) = null$ if and only if $\Gamma_2(x) = null$, and*
   (b) *if $o_i \in \Gamma_i(\Pi)$ and $ref(\Gamma_i(x)) = ref(o_i)$, then $type(\Gamma_1(x)) = type(\Gamma_2(x))$ and $o_1 = o_2$.*

For state Γ and a subset $V \subseteq VAR$, $\Gamma(\Pi{\downarrow_V})$ projects Π onto *V* and it is defined as follows:

1. if $x : C \in V$, $C \in cname$, $o \in \Gamma(\Pi)$ and $ref(\Gamma(x)) = ref(o)$, then $o \in \Gamma(\Pi{\downarrow_V})$
2. if $o \in \Gamma(\Pi{\downarrow_V})$ and a is an attribute of $type(o)$ with a class type, $o_1 \in \Gamma(\Pi)$ and $ref(o.a) = ref(o_1)$, then $o_1 \in \Gamma(\Pi{\downarrow_V})$
3. $\Gamma(\Pi{\downarrow_V})$ only contains objects constructed from $\Gamma(\Pi)$ and the values of the external variables following the above two rules.

In particular, when we restrict a state Γ on the external variables *glb* and projects Π onto these variables, we obtain an *external state* in which all objects in the system configuration are attached to variables. Therefore, the restriction plays the role of *garbage collection*.

For a given state, each expression e, $visible(e)$ is true if and only if one of the following conditions holds:

1. e is a declared simple variable $x \in$ *var*, or
2. $e \equiv$ *self*.a and there exists a class name $N \in$ *cname* such that $N \succeq$ *atype*(*self*) and $N.a \in$ *visibleattr*, or
3. e is of the form $e_1.a$ and e_1 is not *self* such that *visible*(e_1), there exists a $N \succeq$ *type*(e_1) and $N.a \in$ *visibleattr*.

Condition (2) says that if *type*(*self*) is *C* and *atype*(*self*) is *D*, then the attributes of *D* can be accessed in the method bodies of the methods *D* which are inherited or over rewritten from the casted class *C*. Condition (3) ensures an attribute of an object other than *self* can be directly accessed if and only if it is an attribute in the casted type, i.e. the type of the expression itself. This would become clearer after understanding the semantics of a method invocation.

3.3 Evaluation of Expressions

The evaluation of an expression e under a given state determines its type *type*(e) and its value that is a member of *type*(e) if this type is a built-in primitive type, otherwise a value in *REF* $\times$ *CNAME*. The evaluation makes use of the system configuration. An expression can only be evaluated when it is well-defined. Some well-definedness conditions are static that can be checked at compiling time, but some are dynamic. The evaluation results of expressions are given in Table. 2, where we only give an example (at the bottom of the table) about well-defined expression on built-in primitive types.

Notice the definition of type casting $(C)e$ requires that the base type e be a subclass of *C*. This is implemented in Java by the testing command $C.class.isInstance(e)$. This covers all the following both casting up when the casted type *type*(e) is a subclass of *C* too, and casting down when *type*(e) is superclass of *C*.

Semantics of Commands. A typical aspect of an execution of an object-oriented program is about how objects are to be attached to program variables (or entities [38]). An attachment is made by an assignment, the object creation of an object or passing a parameter in a method invocation. With the approach of UTP, these different cases are unified as an assignment of a value to a program variable. Also, all other programming constructs will be defined in exactly the same way as their counter-parts in a procedural language. We only define the commands which are typical for object-orientation and the definition for the other commands remains same as in the imperative programming as we introduced in Section 2, provided they are well-defined. The semantics $[\![c]\!]$ of each command c has its well-defined condition $\mathcal{D}(c)$ as part of its precondition and thus has the form of $\mathcal{D}(c) \Rightarrow (p \vdash R)$ or $\mathcal{D}(c) \wedge p \vdash R$.

Assignments. An assignment *le* $:= e$ is well-defined if both *le* and e are well-defined and current type of e matches the declared type of *le*

$$\mathcal{D}(\mathit{le} := e) \stackrel{def}{=} \mathcal{D}(\mathit{le}) \wedge \mathcal{D}(e) \wedge (\mathit{type}(e) \in \mathit{cname} \Rightarrow (e = \mathit{null}) \vee (\mathit{type}(e) \preceq \mathit{dtype}(\mathit{le})))$$

Notice that the well-definedess checking here includes dynamic type matching. However, for a language with strong typing, the strong static typing condition would be

Table 2. Evaluation of Expressions

Expression	Evaluation
null	$\mathcal{D}(\mathit{null}) \stackrel{def}{=} \mathit{true}, \quad \mathit{type}(\mathit{null}) \stackrel{def}{=} \mathit{NULL}, \quad \mathit{ref}(\mathit{null}) \stackrel{def}{=} \mathit{null}$
x	$\mathcal{D}(x) \stackrel{def}{=} \mathit{visible}(x) \wedge (\mathit{dtype}(x) \in \mathcal{T} \vee \mathit{dtype}(x) \in \mathit{cname})$ $\wedge \ \mathit{dtype}(x) \in \mathcal{T} \Rightarrow \mathit{head}(\overline{x}) \in \mathit{dtype}(x)$ $\wedge \ \mathit{dtype}(x) \in \mathit{cname} \Rightarrow$ $\mathit{ref}(\mathit{head}(\overline{x})) \in \mathit{ref}(\Pi(\mathit{dtype}(x)))$ $\mathit{type}(x) \stackrel{def}{=} \begin{cases} \mathit{dtype}(x) & \mathit{dtype}(x) \in \mathcal{T} \\ \mathit{type}(\mathit{head}(\overline{x})) & \text{otherwise} \end{cases}$
$le.a$	$\mathcal{D}(le.a) \stackrel{def}{=} \mathcal{D}(le) \wedge le \neq \mathit{null}$ $\wedge \ \mathit{dtype}(le) \in \mathit{cname} \wedge \mathit{visible}(le.a)$ $\mathit{type}(le.a) \stackrel{def}{=} \mathit{type}(\mathit{state}(le)(a))$ $\mathit{ref}(le.a) \stackrel{def}{=} \mathit{ref}(\mathit{state}(le)(a))$
$(C)e$	$\mathcal{D}((C)e) \stackrel{def}{=} \mathcal{D}(e) \wedge \mathit{type}(e) \notin \mathcal{T} \wedge \mathit{atype}(e) \preceq C$ $\mathit{type}((C)e) \stackrel{def}{=} C$ $\mathit{ref}((C)e) \stackrel{def}{=} \mathit{ref}(e)$
e/f	$\mathcal{D}(e/f) \stackrel{def}{=} \mathcal{D}(e) \wedge \mathcal{D}(f) \wedge \mathit{dtype}(e) = \texttt{Real}$ $\wedge \ \mathit{dtype}(f) = \texttt{Real} \wedge \mathit{value}(f) \neq 0$ $\mathit{value}(e/f) \stackrel{def}{=} \mathit{value}(e)/\mathit{value}(f)$

enough $\mathit{dtype}(e) \preceq \mathit{dtype}(le)$, as it implies $\mathit{type}(e) \preceq \mathit{dtype}(le)$. Also, together with the well-definedness $\mathcal{D}(e)$, when e is an object $\mathcal{D}(le := e)$ ensures that $\mathit{atype}(e) \preceq \mathit{dtype}(le)$.

There are two cases of assignment. The first is to (re-)attach a value to a variable (i.e. change the current value of the variable), but this can be done only when the type of the object is consistent with the declared type of the variable. The attachment of values to other variables are not changed.

$$[\![x{:=}e]\!] \stackrel{def}{=} \{x\} : \mathcal{D}(x{:=}e) \vdash (\overline{x}' = \langle \mathit{value}(e) \rangle \cdot \mathit{tail}(\overline{x}))$$

As we do not allow attribute hiding or redefinition in subclasses, the assignment to a simple variable has not side-effect, and thus the Hoare triple

$$\{o_2.a = 3\}\ o_1 := o_2\ \{o_1.a = 3\}$$

is valid in our model, where $o_1 : C_1$ and $o_2 : C_2$ are variables, $C_2 \preceq C_1$ and $a : \texttt{Int}$ is protected attribute of C_1. This has made the theory simpler than the Hoare-Logic based semantics for object-oriented programming in [41].

The second case is to modify the value of an attribute of an object attached to an expression. This is done by finding the attached object in the system configuration Π and modifying its state accordingly. Thus, all variables attached to the reference of this object will be updated.

$$[\![le.a := e]\!] \stackrel{def}{=} \{\Pi(\mathit{dtype}(le))\} : \mathcal{D}(le.a{:=}e) \vdash \left(\begin{array}{l} \Pi(\mathit{dtype}(le))' = \Pi(\mathit{dtype}(le)) \uplus \\ \{o \oplus \{a \mapsto \mathit{value}(e)\} \mid o \in \Pi \wedge \mathit{ref}(o) = \mathit{ref}(le)\} \end{array} \right)$$

For example, let x be a variable of type C such that C has an attribute d of D and D has an attribute a of integer type. $x.d.a := 4$ will change state of $x = \langle r_1, C, \{d \mapsto r_2\}\rangle$, where reference r_2 is the identity of $\langle r_2, D, \{a \mapsto 3\}\rangle$ to $x = \langle r_1, C, \{d \mapsto r_2\}\rangle$, but the r_2 is now the identity of the object $\langle r_2, D, \{a \mapsto 4\}\rangle$.

This semantic definition shows the side-effect of an assignment and does reflect the object-oriented feature pointed out by Broy in [6] that an invocation to a method of an object which contains such an assignment or an instance creation defined later on, changes the system configuration Π.

Law 1. $(le := e; le := f(le)) = (le := f(e))$

Law 2. $(le_1 := e_1; le_2 := e_2) = (le_2 := e_2; le_1 := e_1)$*, provided* le_1 *and* le_2 *are distinct simple names which do not occur in* e_1 *or* e_2*.*

Note that the law might not be valid if le_i are composite expressions. For instance, the following equation is not valid when x and y have the same reference:

$$(x.a := 1;\ y.a := 2) = (y.a = 2;\ x.a = 1)$$

Object Creation. The $C.new(le)$ is well-defined if

$$C \in cname \wedge \mathcal{D}(le) \wedge dtype(le) \succeq C$$

The command creates a new object, attaches the object to x and set the initial values of the attributes to the attributes of x too.

$$[\![C.new(le)]\!] \stackrel{def}{=} \{le, \Pi(C)\}: \\ D(C.new(le)) \vdash \exists r \notin ref(\Pi) \cdot (AddNew(C, r) \wedge Modify(le))$$

where

$$AddNew(C, r) \stackrel{def}{=} \Pi(C)' = \Pi(C) \\ \cup \{\langle r, C, \{a_i \mapsto init(C.a_i)\}\rangle \mid a_i \in AAttr(C)\}$$

$$Modify(le) \stackrel{def}{=} \begin{pmatrix} le \in localvar \wedge \overline{le}' = \langle r, C\rangle \cdot tail(\overline{le}) \wedge \\ TypeSeq'(le) = \langle C\rangle \cdot tail(TypeSeq(le)) \\ \vee\ le \notin localvar \wedge \{le\} : true \vdash (le' = (r, C)) \end{pmatrix}$$

Here we assume if $dtype(C.a_i) = M$, the assignment $a_i \mapsto init(C.a_i)$ is $a_i \mapsto M.new(C.a_i)$.

For creation of objects, we have the following laws

Law 3. $C_1.new(x); C_2.new(y) = C_2.new(y); C_1.new(x)$*, provided* x *and* y *are distinct.*

Law 4. *If* x *is not free in the Boolean expression* b*, then*

$$C.new(x); (P \lhd b \rhd Q) = (C.new(x); P) \lhd b \rhd (C.new(x); Q)$$

Local Variable Declaration and Undeclaration. Command $\texttt{var}\ T\ x = e$ declares a variable and initialises it:

$$[\![\texttt{var}\ T\ x = e]\!] \stackrel{def}{=} \{x\} : \mathcal{D}(\texttt{var}\ T\ x = e) \vdash \\ (\overline{x}' = \langle value(e)\rangle \cdot \overline{x}) \wedge TypeSeq'(x) = \langle T\rangle \cdot TypeSeq(x)$$

where

$$\mathcal{D}(\texttt{var}\ T\ x = e) \stackrel{def}{=} (x \in \mathit{localvar}) \wedge \mathcal{D}(e) \wedge \mathit{type}(e) \notin \mathcal{T} \Rightarrow \mathit{type}(e) \preceq T$$

We define $[\![\texttt{var}\ T\ x]\!] \stackrel{def}{=} \sqcap_{d \in T} [\![\texttt{var}\ T\ x = d]\!]$.

Command `end` terminates the block of permitted use a variable:

$$[\![\texttt{end}\ x]\!] \stackrel{def}{=} \{x\}{:}\mathcal{D}(\texttt{end}\ x) \vdash \overline{x}' = \mathit{tail}(\overline{x}) \wedge \mathit{TypeSeq}'(x) = \mathit{tail}(\mathit{Tseq}(x))$$

where $\mathcal{D}(\texttt{end}\ x) \stackrel{def}{=} x \in \mathit{localvar}$.

Declaration and undeclaration distribute over conditional choice.

Law 5. *If x is not free in b, then*

$$\texttt{var}\ T\ x = e; (P \lhd b \rhd Q) = (\texttt{var}\ T\ x = e; P) \lhd b \rhd (\texttt{var}\ T\ x = e; Q)$$
$$\texttt{end}\ x; (P \lhd b \rhd Q) = (\texttt{end}\ x; P) \lhd b \rhd (\texttt{end}\ x; Q)$$

Initialisation becomes void if the declared variable is updated immediately.

Law 6. $(\texttt{var}\ T\ x = e; x := f) \sqsubseteq \texttt{var}\ T\ x = f$

Note that the two commands in the above law are not equivalent it is possible that e is not well-defined.

Assignment to a variable just before the end of its scope is irrelevant if it is well-defined.

Law 7. $(x := e;\ \texttt{end}\ x) \sqsubseteq \texttt{end}\ x$

Both declaration and undeclaration are commutative.

Law 8. $(\texttt{var}\ T_1\ x = e_1; \texttt{var}\ T_2\ y = e_2) = (\texttt{var}\ T_2\ y = e_2; \texttt{var}\ T_1\ x = e_1)$, *provided y is not in e_1 and x does not appear in e_2.*

Law 9. $(\texttt{end}\ x; \texttt{end}\ y) = (\texttt{end}\ y; \texttt{end}\ x)$

Law 10. $(\texttt{var}\ T\ x = e; \texttt{end}\ y) = (\texttt{end}\ y; \texttt{var}\ T\ x = e)$, *provided y is not in e.*

Method Call. For a method signature $m(T_1\ x; T_2\ y; T_3\ z)$, let *ve*, re and *vre* be lists of expressions. Command $le.m(ve; re; vre)$ is well-defined if *le* is well-defined and it is a non-null object such that a method $m \mapsto (T_1\ x; T_2\ y; T_3\ z, c)$ is in the casted type $\mathit{type}(le)$ of *le*:

$$\begin{aligned}\mathcal{D}(le.m(ve; re; vre)) \stackrel{def}{=}\ & \mathcal{D}(le) \wedge \mathit{type}(le) \in \mathit{cname} \wedge (le \neq \mathit{null}) \\ & \wedge N \in \mathit{cname} \cdot N \succeq \mathit{type}(le) \\ & \wedge \exists(m \mapsto (T_1\ x; T_2\ y; T_3\ z, c)) \in \mathit{op}(N)\end{aligned}$$

The execution of this method invocation assigns the values of the actual parameters *v* and *vr* to the formal value and value-result parameters of the method *m* of the object *o* that le refers to, and then executes the body of m under the environment of the class owning method $m()$. After it terminates, the value of the result and value-result parameters of *m* are passed back to the actual parameters *r* and *vr*.

$$\begin{aligned}[\![le.m(ve; re; vre)]\!] \stackrel{def}{=}\ & (\mathcal{D}(le.m(ve; re; vre)) \Rightarrow \\ & \exists C \in \mathit{cname} \cdot (\mathit{atype}(le) = C) \\ & \wedge \left(\begin{array}{l} [\![\texttt{var}\ T_1\ x = ve, T_2\ y, T_3\ z = vre]\!]; \\ [\![\texttt{var}\ C\ \mathit{self} = le]\!]; \\ [\![\mathit{Execute}(C.m)]\!];\ [\![re, vre := y, z]\!]; \\ [\![\texttt{end}\ \mathit{self}, x, y, z]\!] \end{array}\right)\end{aligned}$$

where *Execute*(*M.m*) sets the execution environment, then executes the body and reset the environment afterwards. There are the following cases:

Case 1: If $m(T_1\ x; T_2\ y; T_3\ z)$ is not declared in C but in a superclass of C, i.e. there exists a command c such that $(m \mapsto (T_1\ x; T_2\ y; T_3\ z, c)) \in op(N)$ for some $N \succeq C$, then

$$Execute(C.m) \stackrel{def}{=} Ex_C(superclass(C).m)$$

where if $m() \in op(M)$ then

$$Ex_C(M.m) \stackrel{def}{=} Set(C, M); SELF_C^M(body(M.m)); Reset$$

else

$$Ex_C(M.m) \stackrel{def}{=} Ex_C(superclass(M).m)$$

Case 2: If $m(T_1\ x; T_2\ y; T_3\ z)$ is declared in class C itself, that is for some command c $(m \mapsto (T_1\ x; T_2\ y; T_3\ z, c)) \in op(C)$, then

$$Execute(C.m) \stackrel{def}{=} Set(C, C); SELF_C^C(body(C.m)); Reset$$

where

- $body(C.m)$ is the body c of the method being called.
- The design $Set(C, M)$ finds out all attributes visible to class M in order for the invocation of method m of M to be executed properly, whereas *Reset* resets the environment to be the set of variables that are accessible to the main program only:

$$Set(C, C) \stackrel{def}{=} \{\texttt{visibleattr}\} : true \vdash$$
$$\texttt{visibleattr}' = \begin{pmatrix} \{C.a \mid a \in pri(C)\} \cup \\ \bigcup_{C \preceq N} \{C.a \mid a \in prot(N) \cup pub(N)\} \cup \\ \bigcup_{N \in pubcname} \{N.a \mid a \in pub(N)\} \end{pmatrix}$$

  and when C and M are different

$$Set(C, M) \stackrel{def}{=} \{\texttt{visibleattr}\} : true \vdash$$
$$\texttt{visibleattr}' = \begin{pmatrix} \{C.a \mid a \in pri(M)\} \cup \\ \bigcup_{M \preceq N} \{C.a \mid a \in prot(N) \cup pub(N)\} \cup \\ \bigcup_{N \in pubcname} \{N.a \mid a \in pub(N)\} \end{pmatrix}$$

$$Reset \stackrel{def}{=} \{\texttt{visibleattr}\} : true \vdash$$
$$\texttt{visibleattr}' = \bigcup_{N \in pubcname} \{N.a \mid a \in pub(N)\}$$

  Set and *Reset* are used to ensure data encapsulation that is controlled by *visibleattr* and the well-definedness condition of an expression.
- The transformation $SELF_C$ on a command is defined in Table 3, which adds a prefix *self* to each attribute and each method in the command. Notice that as a method call may occur in a command that will change the execution environment, therefore after the execution of the nested call is completed the environment needs to be set back to that of C.

Table 3. The Definition of *SELF*

c or e	$SELF^M_C(c)$ or $SELF^M_C(e)$
skip	*skip*
chaos	*chaos*
$c_1 \lhd b \rhd c_2$	$SELF^M_C(c_1) \lhd SELF^M_C(b) \rhd SELF^M_C(c_2)$
$c_1 \sqcap c_2$	$SELF^M_C(c_1) \sqcap SELF^M_C(c_2)$
$\texttt{var}\ T\ x = e$	$T\ \texttt{var}\ x = SELF^M_C(e)$
$\texttt{end}\ x$	$\texttt{end}\ x$
$C.new(x)$	$C.new(SELF^M_C(x))$
$le := e$	$SELF^M_C(le) := SELF^M_C(e)$
$le.m(ve; re; vre)$	$SELF^M_C(le).m(SELF^M_C(ve); SELF^M_C(re); SELF^M_C(vre))$
$m(ve; re; vre)$	$self.m(SELF^M_C(ve); SELF^M_C(re); SELF^M_C(vre))$
$c_1; c_2$	$SELF^M_C(c_1); Set(C, M); SELF^M_C(c_2)$
$b * c$	$SELF^M_C(b) * (SELF^M_C(c); Set(C, M))$
$le.a$	$SELF^M_C(le).a$
$f(e)$	$f(SELF^M_C(e))$
null	*null*
self	*self*
x	$\begin{cases} self.x, & x \in \bigcup_{C \preceq N} Attr(N) \\ x, & otherwise \end{cases}$

Notice that semantics of a method call defines the method binging rules to ensure that

- only a method with a signature declared in the casted type or above the casted type in the inheritance hierarchy can be accessed, and
- method that is executed is the one defined in the lowest position the inheritance hierarchy from the current type of the active object.

We did not introduce the syntax $super.m$ to explicitly indicate the call to a method according to its definition in the superclass. There is no difficulty to introduce $super.m$ and define its semantics accordingly.

Example 1. To illustrate the semantics of a method invocation, we can consider the bank system with the UML class diagram in Figure 1. We define $Execute(C.m)$ for the method *withdraw*() in the classes of current account and saving account *CA* and *SA*. We assume all classes, except for *Bank*, are private classes, and further notice that

1. the body of *withdraw*() in the superclass *Account* is

   $$balance > x \vdash balance' = balance - x$$

2. subclass *SA* inherits *withdraw*() from *Account*, and
3. subclass *CA* overwrites the body of *withdraw*() into

   $$balance := balance - x$$

For class *CA*,

$$\begin{aligned} \mathit{Execute}(\mathit{CA.withdraw}) &= \mathit{Set}(\mathit{CA},\mathit{CA}); \mathit{SELF}^{\mathit{CA}}_{\mathit{CA}}(\mathit{balance} := \mathit{balance} - x); \mathit{Reset} \\ &= \mathit{visibleattr} := \{\mathit{CA.blance}, \mathit{CA.aNo}\}; \\ &\quad \mathit{self.balance} := \mathit{self.balance} - x; \\ &\quad \mathit{visibleattr} := \emptyset \end{aligned}$$

According to the semantics of a method call to $o.\mathit{withdraw}(e)$, where o is an object of *CA*, the execution of this method call first attaches o to *self*, and then executes the method according to the semantics of $\mathit{Execute}(CA.\mathit{withdraw})$ defined above. It shows that the method is executed according to the current type *CA* and the method is the method of the subclass.

For the case of a saving account

$$\begin{aligned} &\mathit{Execute}(\mathit{SA.withdraw}) \\ &= \mathit{Set}(\mathit{SA},\mathit{Account}); \mathit{SELF}^{\mathit{Account}}_{\mathit{SA}}(\mathit{Account.withdraw}); \mathit{Reset} \\ &= \mathit{visibleattr} := \{\mathit{SA.blance}, \mathit{SA.aNo}\}; \\ &\quad \mathit{self.balance} > x \vdash \mathit{self.balance}' = \mathit{self.balance} - x; \\ &\quad \mathit{visibleattr} := \emptyset \end{aligned}$$

Thus, the invocation to a withdraw method of a saving account is executed according to the definition of the method in the superclass *Account*. ♣

Semantics of Object Systems. Having defined the semantics of a class declaration section and a command, we combine them to define the semantics of an object program $(\mathit{Cdecls} \bullet \mathit{Main})$.

Recall that *Main* consists of a set *externalvar* of the external variables with their types and a command c. For simplicity but without loss of expressive power, we assume that any primitive command in c is in one of the following forms:

1. an assignment $x := e$ such that $x \in \mathit{externalvar}$ and e does not contain subexpressions of the form $le.a$. That is, we do not allow direct access to object attributes in the main method.
2. a creation of a new object $C.\mathit{New}(x)$ for a variable $x \in \mathit{externalvar}$,
3. a method call $x.m(ve; re; vre)$, where x is a variable in *externalvar*.

Main is well-defined if the types of all variables in *externalvar* are either built-in primitive types or public classes declared in *pubcname*:

$$\mathcal{D}(\mathit{Main}) \stackrel{def}{=} \bigwedge_{x \in \mathit{externalvar}} (\mathit{dtype}(x) \in \mathit{pubcname} \vee \mathit{dtype}(x) \in \mathcal{T})$$

The semantics of *Main* is then defined to be

$$[\![\mathit{Main}]\!] \stackrel{def}{=} \mathcal{D}(\mathit{Main}) \Rightarrow [\![c]\!]$$

Before *Main* is executed, the well-definedness of the declaration section has to be checked and the local variables have to be initialised to empty sequences. For this we define a design *Init*:

$$\begin{aligned} \mathit{Init} \stackrel{def}{=} \mathcal{D}(\mathit{Cdecls}) \vdash \mathit{visibleattr}' = \emptyset \wedge (\Pi' = \emptyset) \wedge \\ \bigwedge_{x \in \mathit{var}} (\overline{x}' = <> \wedge \mathit{TypeSeq}'(x) = <>) \end{aligned}$$

Definition 4. *The* **semantics of an object program** *Cdecls • Main is defined to be the following sequential composition*

$$[\![Cdecls \bullet Main]\!] \stackrel{def}{=} \exists\, \Omega, \Omega', glb, glb' \cdot ([\![Cdecls]\!]; Init; [\![Main]\!])$$

This definition of the *closed* semantics allows us to hide the internal information in the execution of a program, only observing the relation between the pre-state and post-state of the external variables whose types are built-in primitive types, and the object type information of the external variables whose types are declared as classes. We cannot observe the information of the objects attached to these variables. We have a less abstract definition for the semantics of an object program.

We define the *open semantics* $[\![Cdecls \bullet Main]\!]_o$ of *Cdecls • Main* as

$$\exists \{\Pi\} \gg glb, \{\Pi'\} \gg glb', \Omega, \Omega' \cdot \\ ([\![Cdecls]\!]; Init; [\![Main]\!]; [\![\Pi' := \Pi\downarrow_{externalvar}]\!])$$

The open semantics allows us to observe the full information about the states of external variables. We can insert the command $\Pi' := \Pi\downarrow_{externalvar}$ at any point of the main method without changing the open and close semantics of a program.

Lemma 1. *For any object program $S = Cdecls \bullet Main$ with c as the command in the main method, we have*

1. $[\![S]\!] = \exists \Pi, \Pi' \cdot [\![S]\!]_o$.
2. *If c is of the form $c_1; c_2$, let S_2 be the program which replaces the command c with $c_1; \Pi' := \Pi\downarrow_{externalvar}; c_2$, then $[\![S]\!]_o = [\![S_2]\!]_o$.*
3. *If c is of the form $c_1; b * (c_2; c_3); c_4$, let S_3 be the program which replaces the loop in Main with $b * (c_2; \Pi' := \Pi\downarrow_{externalvar}; c_3)$, then $[\![S]\!]_o = [\![S_3]\!]_o$.*
4. *If c is of the form $c_1; (c_2; c_3) \lhd b \rhd c_4; c_5$, let S_4 be the program which replaces the conditional choice in Main with $(c_2; \Pi' := \Pi\downarrow_{externalvar}; c_3) \lhd b \rhd c_4$, then $[\![S]\!]_o = [\![S_4]\!]_o$.*
5. *If c is of the form $c_1; (c_2; c_3) \sqcap c_4$, let S_5 be the program which replaces command c in Main with $c_1; (c_2; \Pi' := \Pi\downarrow_{externalvar}; c_3) \sqcap c_4$, then $[\![S]\!]_o = [\![S_5]\!]_o$.*

4 Object-Oriented Refinement

We would like the refinement calculus to cover not only the early development stages of requirements analysis and specification but also the later stages of design and implementation. This section presents the results of our exploration on three kinds of refinement:

1. Refinement relation between object systems.
2. Refinement relation between declaration sections (*structural refinement*).

We only present the definitions and some laws. For detailed study with proofs, we refer to the full version of the paper in [16]. The refinement calculus is used in a case study of the development of a Point of Sale Terminal (POST) [36].

4.1 Refinement of Object Systems

We have defined the refinement relation between commands and shown some examples in the previous section. We now define what we mean by a refinement between two object programs and then focus on the structural refinement. The notation of structural refinement is actually an extension to the notion of data refinement [19].

Definition 5. *Let $S_i = Cdecls_i \bullet Main_i$, $i = 1, 2$, be object programs which have the same set of external variables externalvar. S_1 is a* **refinement** *of S_2, denoted by $S_1 \sqsupseteq_{sys} S_2$, if the following implication holds:*

$$\forall externalvar, externalvar', ok, ok' \cdot ([\![S_1]\!] \Rightarrow [\![S_2]\!])$$

Example 2. For any class declaration *Cdecls*, we have the following:

1. $S_1 = Cdecls \bullet (\{x : C\}, C.new(x))$ and $S_2 = Cdecls \bullet (\{x : C\}, C.new(x); C.new(x))$ are equivalent.
2. We assume class $C \in pubcname$, $\langle a : \mathtt{Int}, d\rangle \in attr(C)$, $get(\emptyset; \mathtt{Int}\ z; \emptyset)\{z := a\}$ and $update()\{a := a + c\}$ in $op(C)$, then

$$Cdecls \bullet (\{x : C, y : \mathtt{Int}\}, C.new(x); x.update(); x.get(y))$$

and

$$Cdecls \bullet (\{x : C, y : \mathtt{Int}\}, C.new(x); x.update(); x.get(y); C.new(x))$$

are equivalent.

Proof. We give a proof for item (2) of this example. We denote the first program by S_1 and the second by S_2. Assume the declaration section is well-defined, as otherwise both programs are chaos. Then it is easy to check the main methods are both well-defined. The structural variables Ω are calculated according to the definition. Let d be the initial value of attribute a of C and σ_0 denote the initial state of an object of C when it is created. We calculate the semantics of S_1:

$$\begin{aligned}
&[\![C.new(x); x.update(), x.get(y)]\!] \\
=\ &\begin{pmatrix} true \vdash \exists r \in REF \cdot (\Pi' = \{\langle r, C, \sigma_0\rangle\} \wedge x' = \langle r, C\rangle); \\ [\![x.update(); x.get(y)]\!] \end{pmatrix} \\
=\ &\begin{pmatrix} true \vdash \exists r \in REF \cdot (\Pi' = \{\langle r, C, \sigma_0\rangle\} \wedge x' = \langle r, C\rangle) \wedge \\ self' =<> \wedge \Pi' = \{\langle r, C, \sigma_0 \oplus \{a \mapsto d + c\}\rangle \mid r = ref(x)\}); \\ [\![x.get(y)]\!] \end{pmatrix} \\
=\ &\begin{pmatrix} true \vdash \exists r \in REF \cdot (\Pi' = \{\langle r, C, \sigma_0 \oplus \{a \mapsto d + c\}\rangle\} \wedge \\ \qquad x' = \langle r, C\rangle) \wedge (self' =<>); \\ [\![x.get(y)]\!] \end{pmatrix} \\
=\ &\begin{pmatrix} true \vdash \exists r \in REF \cdot (\Pi' = \{\langle r, C, \sigma_0 \oplus \{a \mapsto d + c\}\rangle\} \wedge \\ \qquad x' = \langle r, C\rangle) \wedge self' =<>); \\ true \vdash self' =<> \wedge z' =<> \wedge y' = d + c \wedge \\ \qquad visibleattr' = \{M.a \mid M \in pubname \wedge a \in pub(M)\} \end{pmatrix} \\
=\ &\begin{pmatrix} true \vdash \exists r \in REF \cdot (\Pi' = \{\langle r, C, \sigma_0 \oplus \{a \mapsto d + c\}\rangle\} \wedge \\ \qquad x' = \langle r, C\rangle) \wedge self' =<> \wedge z' =<> \wedge y' = c + d \wedge \\ \qquad visibleattr' = \{M.a \mid M \in pubname \wedge a \in pub(M)\} \end{pmatrix}
\end{aligned}$$

The semantics $[\![S_1]\!]$ hides Ω, Π, *self* and z by existential quantification. Let $[\![Cdecls]\!]$ be $true \vdash \Omega = \emptyset \wedge \Omega' = \Omega_0$, we have $[\![S_1]\!]$ equals to

$$\begin{array}{l}\exists \left\{ \begin{array}{l} \Omega, \Omega', self, self', z, z', \\ visibleattr, visibleattr' \end{array} \right\} \cdot ([\![Cdecls]\!]; Init; [\![C.new(x); x.update(), x.get(y))]\!]) \\ = true \vdash \exists r \in REF \cdot x' = \langle r, C\rangle \wedge y' = c + d\end{array}$$

The main method of S_2 is the main method of S_1 followed by command $C.new(x)$ and thus its semantics equals

$$\begin{array}{l} [\![C.new(x); x.update(), x.get(y)]\!]; [\![C.new(x)]\!] \\ = \left(\begin{array}{l} true \vdash \exists r \in REF \cdot (\Pi' = \{\langle r, C, \sigma_0 \oplus \{a \mapsto d + c\}\rangle\} \wedge \\ \qquad x' = \langle r, C\rangle) \wedge self' =<> \wedge z' =<> \wedge y' = c + d \wedge \\ \qquad visibleattr' = \{M.a \mid M \in pubname \wedge a \in pub(M)\}; \\ [\![C.new(x)]\!] \end{array} \right) \\ = \left(\begin{array}{l} true \vdash \exists r \in REF \cdot (\Pi' = \{\langle r, C, \sigma_0 \oplus \{a \mapsto d + c\}\rangle\} \wedge \\ \qquad x' = \langle r, C\rangle) \wedge self' =<> \wedge z' =<> \wedge y' = c + d \wedge \\ \qquad visibleattr' = \{M.a \mid M \in pubname \wedge a \in pub(M)\}; \\ true \vdash \exists p \notin ref(\Pi) \cdot \Pi' = \Pi \cup \{\langle p, C, \sigma_0\rangle\} \wedge (x' = \langle p, C\rangle) \end{array} \right) \\ = \left(\begin{array}{l} true \vdash \exists r, p \in REF \cdot ((p \neq r) \wedge \\ \qquad \Pi' = \{\langle p, C, \sigma_0\rangle, \langle r, C, \sigma_0 \oplus \{a \mapsto d + c\}\rangle\} \wedge \\ \qquad x' = \langle p, C\rangle) \wedge self' =<> \wedge z' =<> \wedge y' = c + d \wedge \\ \qquad visibleattr' = \{M.a \mid M \in pubname \wedge a \in pub(M)\} \end{array} \right) \end{array}$$

Hiding the internal variables, $[\![S_2]\!]$ equals

$$true \vdash \exists p \in REF \cdot x' = (p, C) \wedge y' = c + d$$

Thus, we have proved, S_1 and S_2 refines each other.

However, If we change the main methods of these two programs by adding another $x.get(y)$ to the end of both of them. They are not equivalent anymore. The final value of y for the first program remains will be still $d + c$, but for the second one, the final value of y gets the initial value d after the execution. ♣

The discussion at the end of the example shows that program refinement is not quite compositional. In other words, for two main methods, $Main_i = (externalvar, c_i)$, $i = 1, 2$,

$$Cdecls_1 \bullet Main_1 \sqsupseteq_{sys} Cdecls_2 \bullet Main_2$$

does not in general imply

$$Cdecls \bullet (externalvar, c_1; c) \sqsupseteq_{sys} Cdecls \bullet (externalvar, c_2; c)$$

The main reason for this is the global internal variable Π is hidden in the semantics. In fact any program that has internal variables does not have such compositionality.

Theorem 2. *Let Cdecls • Main, C be a public class declared in Cdecls and $Cdecls_1$ be obtained from Cdecls by changing C to a private class. Then if C is not referred in Main,*

$$Cdecls \bullet Main =_{sys} Cdecls_1 \bullet Main$$

where $=_{sys}$ is the equivalence relation $\sqsupseteq_{sys} \cap \sqsubseteq_{sys}$.

The relation $\sqsupseteq_{sys}$ is reflexive and transitive.

4.2 Structure Refinement

The proof in **Example** 2 shows that the local variables and *visibleattr* of a program are constants after each method invocation. When the main methods in the programs are syntactically identical, the relation between their system states is determined by the relation between the structure of these programs, i.e. their class names, attributes, sub-superclass relations, and methods in the classes.

An object-oriented program design is mainly about designing classes and their methods, and a class declaration section can in fact support many different application main programs. The rest of this section focuses on structural refinement.

Definition 6. *Let $Cdecls_1$ and $Cdecls_2$ be two declaration sections. $Cdecls_1$ is a* **refinement** *of $Cdecls_2$, denoted by $Cdecls_1 \sqsupseteq_{class} cdecls_2$, if the former can replace the later in any object system:*

$$Cdecls_1 \sqsupseteq_{class} Cdecls_2 \stackrel{def}{=} \forall Main \cdot (Cdecls_1 \bullet Main \sqsupseteq_{sys} Cdecls_2 \bullet Main)$$

Intuitively, it states that $Cdecls_1$ supports at least the same set of services as $Cdecls_2$. It is obvious that $\sqsupseteq_{class}$ is reflexive and transitive. We use $=_{class}$ to denote the equivalence relation $\sqsupseteq_{class} \cap \sqsubseteq_{class}$. When there is no confusion, we omit the subscript when we discuss about structural refinement.

A structural refinement does not allow to change the main method. So every public class in $Cdecls_2$ has to be declared in the refined declaration section $Cdecls_1$, and every method signature in a public class of $Cdecls_2$ has to be declared in $Cdecls_1$, otherwise there are main methods which are well-defined under $Cdecls_2$ but not under $Cdecls_1$. Also recall that a main method only change objects by method invocations to public classes.

In the full version of rCOS for object systems [16], we have shown how structural refinement between two class declaration sections by *structural transformations* and *upwards* and *downwards simulations* of public class methods. A structural transformation between two declaration sections is actually a transformation between the corresponding UML class diagrams of the declaration sections. In [16], a proof is given to show that the two classes diagrams (class declaration sections) in Figure 2 refine each other if *C* is the only public class.

4.3 Laws of Structural Refinement

The following refinement laws capture the basic principles in object-oriented design and decomposition, and can be used to prove general object-oriented design patterns within the UML framework:

1. Adding a class declaration: this allows us to add a class into the class diagram, sequence diagrams and state machines of the methods of the new class.
2. Introducing a *fresh* private attribute to a class: this corresponds to adding a fresh attribute of a primitive type to the class or adding a directed association from the class to another in the class diagram.
3. Promoting a private attribute of a class to a protected attribute, and a protected attribute to a public attribute: the same refinements can be applied to a class diagram.

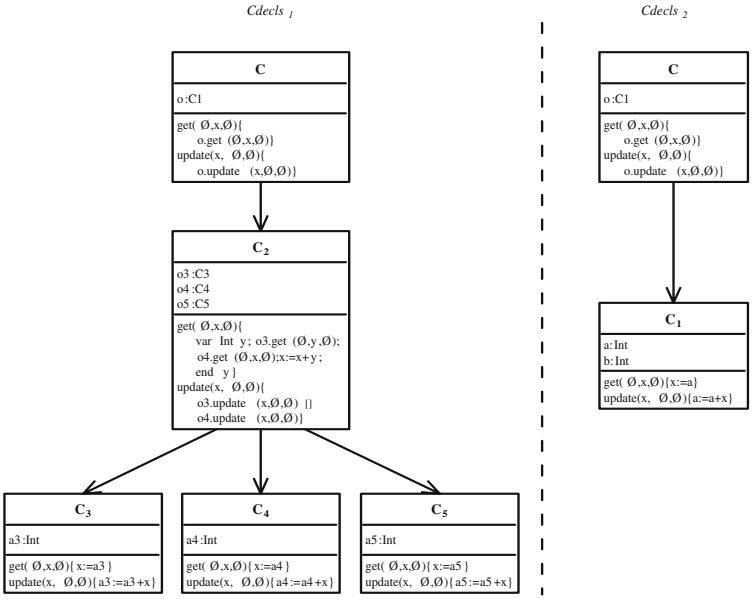


Fig. 2. Example of Structural Refinement

4. Adding a *fresh* method into a class: this allows us to add a method signature into the class in the class diagram, and add a sequence diagram, modify the state machine to incorporate this method. The newly added methods must not violate any state constraint required by the model.
5. Refining the body command of a method $m()\{c\}$ in a class: this leads to the replacement of the subsequence diagrams corresponding to the occurrences of $m()$, and refine the actions of transitions with $m()$ as the triggering event in the state machine of the class.
6. Introducing inheritance: If none of the attribute of class *N* is defined in class *M* or any superclass of *M*, we can make *M* a direct superclass of *N*.
7. Moving some attributes from a class to its direct superclass.
8. Introducing a fresh superclass to a class: If *M* is not in the class declaration, we can introduce M and make it a superclass of an existing class *N*.
9. Moving common attributes of classes which are direct subclasses of a class to the superclass.
10. Moving a method from a class to its direct superclass.
11. Copying (**not** removing) a method of a class to its direct subclass.
12. Removing unused attributes: for a private attribute, it can be removed if it does not appear in any method of the class; for a protected attribute, it can be removed if it does not appear in any method of the class or any of its subclasses; for a public attribute, it can be removed if it does not appear in any method. This is because the main method does not access attributes directly.

We can also refine a class diagram by flattening it into a diagram without inheritance relations between classes. Refinement rules are also available for the object-oriented design patterns. General Responsibility Assignment Software Patterns (GRASP) [25] is a frequently used object-oriented design technique. We have used the facade controller in a requirement specification. One of the most important design patterns is called the *expert pattern*, which shows how part of a functionality of a class can be delegated to another class:

Law 11. (**Expert**) *If a method of a class contains a subcommand that can be realized by a method of another class, we can replace that subcommand with a method invocation to the of the latter class (see Figure 3).*

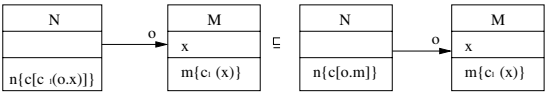


Fig. 3. Expert Pattern

Note that the sequence diagrams and state machines involving $N :: m()$ are refined accordingly. They are not shown here due to the length limit of this paper.

The *Low-Coupling Pattern* of GRASP, on the other hand, can help us remove unnecessary associations to reduce the coupling between classes and simplify reuse and maintenance.

Law 12. (**Low Coupling**) *A call from one class to a method of another can be realized via a third class that is associated with these two classes. This is shown in Figure 4.*

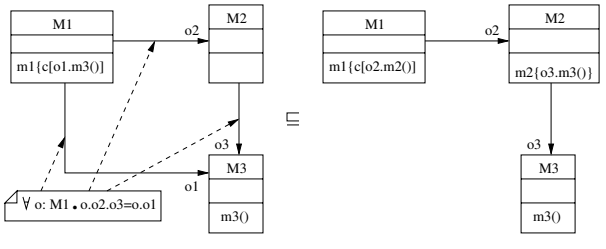


Fig. 4. Low Coupling Pattern

The *High-Cohesion Pattern* corresponds to the principle to decompose a complex class into several related classes. A highly cohesive design makes reuse and maintenance more flexible.

Law 13. (**High Cohesion**) *Assume two methods $m_1()$ and $m_2()$ in a class M and m_1 does not depend on m_2 (though $m_2()$ may call $m_1()$), we can decompose the class into three associated classes so that the original class M only delegates the functionalities to the newly introduced classes. There are two ways of doing this, as shown in Figure 5.*

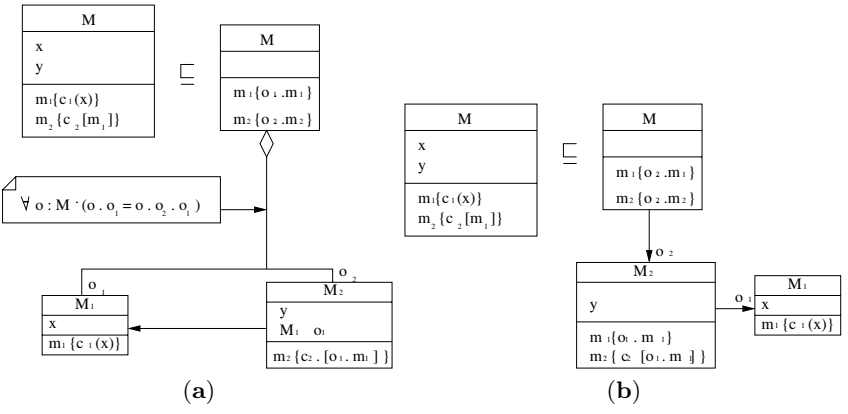


Fig. 5. High Cohesion pattern

The case (a) in Law 13 requires M to be coupled with both M_1 and M_2; and in case (b) M is only coupled with M_2, but more interactions are needed between M_2 and M_1.

The other design patterns in [13], such as *Adaptor Pattern*, *Observer Pattern*, *Strategy Pattern* and *Abstract Factory Pattern* can also be formalized.

In fact the laws above are also reversible and thus can be used for re-engineering. This also implies the result in [5] that every object-oriented program can be converted back to a normal form specification corresponding to an imperative program. Moreover, such a normal form in our framework corresponds to the requirement specification in terms of use cases [26,32]. In [25,29], the five GRASP patterns are systematically used for the development of a case study.

5 Component Systems

Using components to build and maintain software systems is not a new idea. However, it is today's growing complexity of these systems that forces us to turn this idea into practice [45,9,18]. While component technologies such as COM, CORBA, and Enterprise JavaBeans are widely used, there is so far no agreement on standard technologies for designing and creating components, nor on methods for composing them. Finding appropriate formal approaches for specifying components, the architectures for composing them, and the methods for component-based software construction, is correspondingly challenging. In this section, we consider a contract-oriented approach to the specification, design and composition of components. Component specification is essential as it is impossible to manage change, substitution and composition of components if components have not been properly specified.

5.1 Introduction

When we specify a component, it is important to separate different views about the component. From its user's (i.e. external) point of view, a component *Comp* consists

of a set of *provided services* [45]. The syntactic specification of the provided services is described by an interface, defining the operations that the component provides with their signatures. This is also called the *syntactic specification* of a component. COM and CORBA use IDL, and JavaBeans uses Java Programming Language to specify component interfaces. Such a syntactic specification of a component does not provide any information about the effect, i.e. the *functionality* of invoking an operation of a component or the *behavior*, i.e. the temporal order of the interface operations, of the component.

For the functional specification of the operations in an interface, it is however necessary to know the conceptual state of the component. Consequently, the interface specification contains a so-called *information model* [9,11]. In the context of such a model, we still specify an operation m by a *design* $p(x) \vdash R(x, x')$ that is seen as a *contract* between the component and its client [9,18]. This definition of a *contract* also agrees with that of [39,40]. To use the service m, a client has to ensure the pre-condition $p(x)$, and when this is true the component must guarantee the post-condition Q. We then define a *contract* of an interface by associating the interface with a set of *features* that we will call *fields* and assigning each a design $\mathit{MSpec}(m)$ to each interface operation m. The types of the fields are given in a *data/class model*.

The contract for the provided interface of a component allows the user to check whether the component provides the services required by other components in the system, without the need to know the design and implementation of the component. It also commits (or requires) the designers of the component who have to design the component's provided services. A designer of the component under consideration (*CuC*) may decide to use services provided by other components. These services are called *required services* [45] of CuC. Components that provide the required services of CuC can be built by another team or bought as a component-off-the-shelf (COTS). To use a component to assemble a system, one needs to know the specifications of both its provided and required services.

We will specify the design of a component by giving each operation m in the provided interface a program specification text $\mathit{MImpl}(m)$ in rCOS. In *MImpl(m)*, calls to operations in a required interface are allowed. We can then verify whether $\mathit{MImpl}(m)$ refines the specification of m given in a contract of the provided interface. The *verifier* of a component needs to know the contracts of the provided interfaces, the contracts of the required interfaces, and the specification text for each operation m of the provided interface. We can thus understand a component as a relation between contracts of the required interfaces and contracts of the provided interface: given a contract for each required interface, we can calculate a design of an m from $\mathit{MImpl}(m)$ and check whether it conforms to the specification $\mathit{MSpec}(m)$ defined by the contract of the provided interface. A design of a component can be further refined into an implementation by refining the data/class model and then operation specifications $\mathit{MImpl}(m)$.

A component assumes an architectural context defined by its interfaces. We *connect* or *compose* two components Comp_1 and Comp_2 by linking the operations in the provided interface of one component to the matching operations of a required interface of another. For this, we have to check whether the provided interface of component Comp_1 contains the operations of a required interface of component Comp_2, and whether the contract of the provided interface of Comp_1 meets the contract of the required interface of Comp_2.

If $Comp_1$ and $Comp_2$ match well, the composition $Comp_1||Comp_2$ forms another component. The provided interface of $Comp_1||Comp_2$ is the *merge* of the provided interfaces of $Comp_1$ and $Comp_2$. The required interfaces of $Comp_1||Comp_2$ are the union of required interfaces of $Comp_1$ and $Comp_2$, excluding (*by hiding*) the matched interfaces of $Comp_1$ and $Comp_2$. For defining composition, interfaces can be *hidden* and *renamed*.

A component is also replaceable, meaning that the developer can replace one component with another, may be *better*, as long as the new one provides and requests the same services. A component is better than another if it can provide more services, i.e. the contracts for its provided interfaces refine those of the other, with the same required services. Component replaceability is based on the notion of *component refinement*.

In this section we present a model of components that allows us to

- describe and check the syntactic dependency and composability among components in terms of interfaces,
- specify and reason about function composability and substitutability of a component in terms contracts,
- correctness and substitutability of component designs and implementation with respect the contract specification of the component.

We leave the specification, correctness and substitutability of interaction protocols of components in future work.

5.2 Interfaces

An *interface I* is a set of *operation* (or *method*) *signatures* $m(\underline{U}\ \underline{x}; \underline{V}\ \underline{y}; \underline{W}\ \underline{z})$, where m is called the *name* of the operation. An interface can be specified as a family of operation signatures in the following format:

$$
\begin{array}{l}
\textbf{Interface } I \{ \\
\qquad \textbf{Method} : m_1(\underline{U}_1\ \underline{x}_1; \underline{V}_1\ \underline{y}_1; \underline{W}_1\ \underline{z}_1); \\
\qquad\qquad \ldots; \\
\qquad\qquad m_k(\underline{U}_k\ \underline{x}_k; \underline{V}_k\ \underline{y}_k; \underline{W}_k\ \underline{z}_k) \\
\}
\end{array}
$$

Merge Interfaces

It is often the case that there are a number of components, each providing a part of the operations in the required interface of another component. We thus need to *merge* these components to provide one single interface to match the interface required by the other component.

Two interfaces I_1 and I_2 are *composable* provided that every operation name that appears in both I_1 and I_2 must be declared with the same signature. This condition is not too restrictive as to use a component designed for an application in another or specialize a generic component for a special application, renaming or adding a *connector* component [2,42] can be used to *customize* the component.

Definition 7. *Let* $\{I_k : \mid k \in K\}$ *be a finite family of composable interfaces. Their* **merge** $\uplus_{k\in K} I_k$ *is defined by* $\uplus_{k\in K} I_k \stackrel{def}{=} \cup_{k\in K} I_k$.

5.3 Contracts

Only a *syntactic specification* of its interface is not enough for the use or the design of a component. We also need to specify the effect, i.e. the *functionality*, of invoking an interface operation. This requires one to associate the interface to a *conceptual state space*, and a specification of how the states are changed by the operation under certain pre-conditions. We view such a *functional specification* of an interface as a contract between the component *client* and the component *developer*. The contract is the specification of the component that the developer has to implement. The contract is also between a user of the component and a provider of an implementation of the interface: the component has to provide the services promised by the specification *provided* that the user uses the component according to the precondition. To define the conceptual state space of a contract for an interface and the types for the parameters of the interface operations, we assume that a type is either a primitive built-in or a class of objects. This allows our framework to support both imperative and object-oriented programming in the design of a component. The type definitions is given by a class declaration section and it declares a class structure Ω, called the *information model* of the component.

Given an interface *I*, an information model Ω declared by a class declaration section, a set *A* of variable declarations of the form $T\ x$ where T is either a primitive type or a class declared in Ω, called the type of x, we define the alphabet α as the union of sets of the variables, the input and output parameters of the operations of *I*.

$$
\begin{array}{ll}
in\alpha & \stackrel{def}{=} A \cup \{x \in \underline{x} \cup \underline{z} \mid m(\underline{U}\ \underline{x}; \underline{V}\ \underline{y}; \underline{W}\ \underline{z}) \in I\} \\
out\alpha & \stackrel{def}{=} A \cup \{y \in \underline{y} \cup \underline{z} \mid m(\underline{U}\ \underline{x}; \underline{V}\ \underline{y}; \underline{W}\ \underline{z}) \in I\} \\
out\alpha' & \stackrel{def}{=} \{x' \mid x \in out\alpha\} \\
\alpha & \stackrel{def}{=} in\alpha \cup out\alpha
\end{array}
$$

A *conceptual state* for $\langle I, \Omega \rangle$ is a well-typed mapping from the variables α to their value spaces. It is in fact the state space Ξ determined by Ω, plus values of variables in $out\alpha$ of primitive types that is a snapshot of the models consisting the current objects of the classes and links by the associations or attributes that relate these objects, as well as the values of variables of primitive types. As before, a *specification* of an operation $m(\underline{U}\ \underline{x}; \underline{V}\ \underline{y}; \underline{W}\ \underline{z})$ in an alphabet α is a *framed design* $\beta : Spec$.

Definition 8. *A* **contract** *is a tuple* $Contr = (I, \Omega, A, MSpec, Init)$ *where I is an interface,* Ω *is the information model, A is a set of variables, called the* fields *of Contr, whose types are either declared in* Ω *or primitive types, and MSpec a function that maps each operation of I to a specification, and Init an initial condition that defines some values to fields as their initial values.*

If no field is of an object type, we will omit the information model from the specification of a contract. In modular programming, a primitive contract is a specification of a module that defines the behavior of the operations in its interface. However, later we will see that contracts can be *merged* to form another contract and this corresponds to the merge of a number of modules. In object-oriented programming, a primitive contract specifies an initialized class, i.e. an object, whose public methods are operations in the interface. This class *wraps* the classes in the information model Ω, and provides the

interface operations to the environment. In the Java-like rCOS syntax, such a contract can be written as

Interface I {**Meth** : $\{m() \mid m() \in I\}$};
Cdecls;
Class C implements I {**Attr** : $A = Init$;
Method : $\{m()\{MSpec(m)\} \mid m \in I\}$;
main()$\{C.New(x)\}$
}

where **main** provides the condition *Init* when creating the new object of *C* attached to x with the initial values of the attributes in *A*.

Contracts of interfaces can be merged only when their interfaces are composable and the specifications of the common methods are *consistent*. This merge will be used to calculate the provided and required services when components are composed.

Definition 9. *Contracts* $(I_i, \Omega_i, A_i, MSpec_i, Init_i)$, $i = 1, 2$, *are* **consistent** *if*

1. I_1 *and* I_2 *are composable.*
2. *If* x *is declared in both* A_1 *and* A_2*, it has the same type; and* $Init_1(x) = Init_2(x)$.
3. *Any class name C in both* Ω_1 *and* Ω_2 *has the same class declaration in them.*
4. $MSpec_1(m) \Leftrightarrow MSpec_2(m)$ *for all* $m \in I_1 \cap I_2$.

This definition can be extended to a finite family of contracts.

Definition 10. *Let* $\{Contr_k = (I_k, \Omega_k, A_k, MSpec_k, Init_k)\}$ *be a consistent finite family of contracts. Their* **merge***, (denoted by* $\|_{k \in K}\ Contr_k$*), is defined by*

$$I \stackrel{def}{=} \uplus_k I_k, \qquad \Omega \stackrel{def}{=} \otimes_k \Omega_k, \qquad A \stackrel{def}{=} \otimes_k A_k,$$
$$Init \stackrel{def}{=} \otimes_k Init_k,\ MSpec \stackrel{def}{=} \otimes_k MSpec_k$$

where $\otimes$ *denotes the overriding operator, e.g.* $(MSpec_k \otimes MSpec_{k+1})(m) = MSpec_{k+1}(m)$ *if* $m \in I_k \cap I_{k+1}$*;* $MSpec_k(m)$ *if* $m \in I_k$ *but* $m \notin I_{k+1}$*;* $MSpec_{k+1}(m)$ *otherwise.*

A merge of a family of contracts corresponds the construction of a conceptual model from the partial models of the application domain in the contracts. There are three cases about the partial models:

1. The contracts do not share any fields or modelling elements in their conceptual models. In this case, the system formed by the components of these contracts are most loosely coupled. All communications are via method invocations. Such a system is easy to design and maintain. Composing these components is only plug-in composition.
2. The contracts may share fields, but their conceptual models do not share any common model elements. In this case, application domain is partitioned by the conceptual models of these contracts. And components of the system are also quite loosely coupled and easy to construct and maintain. When composing these components, some simple wiring is needed.

3. The contracts share common model elements in their conceptual models. The refinement/design of the contracts has to preserve the consistency and integrity, generally specified by state invariants, of the model. The more elements they share, the more tightly the components are coupled and the more wiring is needed when composing these components.

Definition 11. *We say that a contract* $Contr_1 = (I_1, \Omega_1, A_1, MSpec_1, Init_1)$ *is* **refined** *by* $Contr_2 = (I_2, \Omega_2, A_2, MSpec_2, Init_2)$*, denoted by* $Contr_1 \sqsubseteq Contr_2$*, if there is a mapping* ρ *from* A_1 *to* A_2 *satisfying*

1. *The initial state is preserved:* $(\underline{x} := Init_1(\underline{x}); \rho) \sqsubseteq (\rho; \underline{y} := Init_2(\underline{y}))$*, where* $\underline{x}$ *is the list of variables defined in* A_1*, and* $\underline{y}$ *the list of variables in* A_2*.*
2. *The behavior of the operations of* $Contr_1$ *are preserved: every operation* m *declared in* I_1 *is also declared in* I_2 *and* $(MSpec_1(m); \rho) \sqsubseteq (\rho; MSpec_2(m))$*.*

The refinement relation between contracts will be used to define component refinement. The state mapping ρ allows that a component developed in an application domain can be used in another application domain if such a mapping can be found.

Theorem 3. *Contract refinement enjoys the properties of program refinement.*

1. $\sqsubseteq$ *is reflexive and transitive and a pre-order.*
2. *(***An upper bound condition***) The merge of a family of contracts refines any contract in the family.*
3. *(***A monotonicity condition***) The refinement relation is preserved by the merge operation on contracts. That is for two consistent families of contracts* $\{Contr_k^i \mid k \in K\}$*,* $i = 1, 2$*. If they do have shared fields and* $Contr_k^1 \sqsubseteq Contr_k^2$ *for all* $k \in K$*, then we have* $\|_{k \in K} Contr_k^1 \sqsubseteq \|_{k \in K} Contr_k^2$*.*

5.4 Component

A component consists of a provided interface and optionally a required interface, and an executable code which can be coupled to the codes of other components via their interfaces.

Definition 12. *A* **component** *Comp is a tuple* $< O, I, \Omega, A, MImpl, Init, R >$ *where*

- O *is an interface, called the* provided *or (*output*)* interface *of Comp.*
- I *is an interface disjoint from* O*, called the* internal interface *of Comp*
- Ω *is an information model*
- A *is a set of fields whose types are all declared in* Ω*.*
- *MImpl maps each operation declared in* $O \cup I$ *to a pair* (α, Q)*, where* Q *is a command written in* rCOS*, and* α *is the alphabet obtained from* A *and the input and output parameters of the operations in* $O \cup I$*.*
- R *is the interface that is disjoint from* O *and* I *and consists of the* operations *(not methods of classes in* Ω*) which are referenced in the program text MImpl*(m) *and bodies of methods of classes in* Ω *but not in* $O \cup I$*, where* $m \in O \cup I$*. R is called the* input *or* required *interface of Comp.*

We call $Contr = (O, I, \Omega, A, MImpl, Init)$ *a* **generalized contract***, as it has internal operations and MImpl provides the specification of each operation of O in terms a general* rCOS *command.*

We will use the 4-tuple $(Contr, I, O, R)$ to denote a component, where *Contr* is a generalized contract for the interface $O \uplus I$.

A contract for R is called a *required services* of the component and a contract of the interface O a *provided services*. Operations in R can be seen as *holes* in the component where their specifications or implementation given in other components that are to be plugged in. Therefore, the provided services of a component depends on its required services plugged in from other components. This leads to the definition of our semantics of a component.

In the above definition, we introduced private operations so that we can hide an output operation by making it a private operation. This will keep the definition *MImpl* valid as the hidden operations may be called in $MImpl(m)$. Hiding interface operations allows to offer different services to different clients.

Definition 13. *(***Hiding***) Let* $Contr = (O, I, A, \Omega, MImpl, Init)$ *be a general contract, and* $H \subseteq O$ *a set of operations. The notation* $Contr \backslash H$ *represents the contract*

$$(O \setminus H, I \cup H, \Omega, A, MImpl, Init)$$

where $S \backslash S_1$ *is set-subtraction.*

Theorem 4. *The hiding operator enjoys the following properties.*

1. $(Contr \backslash H) \sqsubseteq Contr$.
2. $Contr \backslash \emptyset = Contr$.
3. $Contr \backslash H = Contr \backslash (H \cap O)$, *where I is the interface of Contr.*
4. $(Contr \backslash H_1) \backslash H_2 = Contr \backslash (H_1 \cup H_2) = (Contr \backslash H_2) \backslash H_1$
5. $(\|_{k \in K} Contr_k) \backslash H = \|_{k \in K} (Contr_k \backslash H)$

5.5 Semantics Components

Definition 14. *The* **semantics** *of a component Comp is defined as a binary relation between its required services and their corresponding provided services*

$$[\![Comp]\!](Contr_R, Contr'_O) \stackrel{def}{=} (Contr_R >> Comp) \sqsubseteq Contr'_O$$

where the variable $Contr_R$ *takes an arbitrary required service as its value,* $Contr'_O$ *takes a provided service for O, and the notation* $Contr_R >> Comp$ *denotes the provided service*

$$(O, F(\Omega), A, MSpec, Init)$$

where $F(\Omega)$ *is the class model obtained from* Ω *by removing the methods of its classes, and mapping MSpec is defined from the given required service*

$$Contr_R = < R, \Omega_R, A_R, MSpec_R, Init_R >$$

by the recursive equations $MSpec(m) = \mathcal{M}(MImpl(m))$*, where* $\mathcal{M}$ *replaces every call of* $m(inexp, outvar)$ *with the actual input parameters* $inexp$*, output parameters* $outvar$ *and value-result parameters* $vrexp$ *of* O *by its corresponding specification.*

$$\mathcal{M}(m(inexp; outvar; vrexp)) \stackrel{def}{=} \left(\begin{array}{l} \textbf{var}\ T_1\ x = inexp, T_2\ y = outvar, T_3\ z = vrexp; \\ MSpec_R(m); outvar, vrexp := y, z; \\ \textbf{end}\ x,\ y,\ z \end{array}\right)$$
$$\textit{if} \quad m(T_1\ x; T_2\ y; T_3\ z) \in R$$

$$\mathcal{M}(m(inexp; outvar; vrexp)) \stackrel{def}{=} \left(\begin{array}{l} \textbf{var}\ T_1\ x = inexp, T_2\ y = outvar, T_3\ z = vrexp; \\ MImpl(m); outvar, vrexp := y, z; \\ \textbf{end}\ x,\ y,\ z \end{array}\right)$$
$$\textit{if} \quad m(T_1\ x; T_2\ y; T_3\ z) \in O \cup I$$

$$\mathcal{M}(v := e) \stackrel{def}{=} v := e$$

$$\mathcal{M}(\mathcal{F}(\mathsf{c})) \stackrel{def}{=} \mathcal{F}(\mathcal{M}(\mathsf{c})) \quad \textit{for any comand c and context } \mathcal{F}$$

Notice that when a component *Comp* has an empty set of required interface operations, *Comp* is a *closed component* and the notation $Contr_{\emptyset} >> Comp$ becomes a constant that is the semantics of the closed program *Comp*.

For a given contract $Contr_R$ for the required interface of *Comp*, $Contr_R >> Comp$ is a closed component. Let $Contr_O$ be a contract of the provided interface of *Comp* which serves as the specification of the component. We say that *Comp correctly realizes* or *implements* $Contr_O$ with a given required service $Contr_R$ if $Contr_O \sqsubseteq (Contr_R >> Comp)$.

In a modular programming paradigm, a component can be designed and implemented as a module in which each of the operations in the output interface is "programmed" using procedures or functions that are defined either locally in the module or externally in other modules. In this case, the external modules that the component calls methods from must be declared, as well as the types of the attribute values and parameters of its methods. Therefore, a component is in fact not a single module, but an artifact that contains all these declared types and modules. In an object-oriented paradigm, such as Java, a component can be seen as a class that implements the interfaces in O. Including the notation for interfaces and contracts in rCOS, the language provides a formal model for components and the calculus of contract refinement and component refinements.

5.6 Refinement and Composition of Components

For a component *Comp* with provided and required interfaces O and R, the semantics $[\![Comp]\!]$ is a binary relation between the input services and output services.

Theorem 5. ***(Monotonicity and Upwards Closure* [44]*)*** *Let* $Comp =< Contr, I, O, R >$ *and* $\sqsubseteq_R$ *and* $\sqsubseteq_O$ *are the refinement relations among contracts of* R *and among contracts of* O *respectively. Then* $\sqsubseteq_R \circ [\![Comp]\!] \circ \sqsubseteq_O \ = \ [\![Comp]\!]$*, where* $\circ$ *denotes relational composition.*

Thus, for any required services $Contr_R \sqsubseteq Contr'_R$, and provided services $Contr_O \sqsubseteq Contr'_O$, then

$$[\![Comp]\!](Contr_R, Contr'_O) \Rightarrow [\![Comp]\!](Contr'_R, Contr_O)$$

A component $Comp_1$ is a *refinement* of a component $Comp_2$, denoted by $Comp_2 \sqsubseteq Comp_1$, if $Comp_1$ is a sub-relation of $Comp_2$.

Definition 15. *Component $Comp_1$ is a* **refinement** *of $Comp_2$ if $R_1 = R_2 \wedge O_1 = O_2$ and for any required service $Contr_{R_1}$,*

$$(Contr_{R_1} >> Comp_2) \sqsubseteq (Contr_{R_1} >> Comp_1)$$

We therefore have when $Comp_1$ refines $Comp_2$, then for any given required service $Contr_R$ and a contract a provided service $Contr_O$ as the specification, $Comp_1$ realizes $Contr_O$ with $Contr_R$ if $Comp_2$ realizes $Contr_O$ with $Contr_R$.

Definition 16. *Let $Comp_i = (Contr_i, I_i, O_i, R_i)$, $i = 1, 2$, be two components with contracts $Contr_i = (O_i \cup I_i, \Omega_i, A_i)$. Assume that $I_1 \cap I_2 = \emptyset$, $O_1 \cap O_2 = \emptyset$ and $R_1 \cap R_2 = \emptyset$. The* **composition** *$Comp_1 || Comp_2$ is defined to merge their contracts, output interfaces and input interfaces, and to remove those input interfaces of each component that are matched by the output interfaces in another:*

$$Comp_1 || Comp_2 \stackrel{def}{=} < Contr_1 \| Contr_2, I_1 \cup I_2, O_1 \uplus O_2, R_1 \backslash O_2 \cup R_2 \backslash O_1 >$$

Let $I \stackrel{def}{=} I_1 \cup I_2$, $R \stackrel{def}{=} R_1 \backslash O_2 \cup R_2 \backslash O_1$ and $O \stackrel{def}{=} O_1 \cup O_2$. The composition of $Comp_1$ and $Comp_2$ is defined by

$$\begin{aligned}
[\![Comp_1 \| Comp_2]\!](Contr_R, Contr'_O) \stackrel{def}{=} \; & \exists Contr_{R_1}, Contr'_{O_1}, Contr_{R_2}, Contr'_{O_2} \bullet \\
& [\![Comp_1]\!](Contr_{R_1}, Contr'_{O_1}) \wedge \\
& [\![Comp_2]\!](Contr_{R_2}, C'_{O_2}) \wedge \\
& Contr_{R_1} \backslash (R_1 \backslash O_2) = Contr'_{O_2} \backslash (O_2 \backslash R_1) \wedge \\
& Contr_{R_2} \backslash (R_2 \setminus O_1) = Contr'_{O_1} \backslash (O_1 \setminus R_2) \wedge \\
& Contr_R = Contr_{R_1} \backslash (R_1 \backslash O_2) \| Contr_{R_2} \backslash (R_2 \backslash O_1) \wedge \\
& Contr'_O = Contr'_{O_1} \backslash (R_2 \backslash O_1) \| Contr'_{O_2} \backslash (R_1 \backslash O_2)
\end{aligned}$$

This definition allows an output interface and thus part of provided service of one component to be shared among a number other components. Hiding can be used to *internalize* the part of a provided service of one component that is used in another component: $(Comp_1 \| Comp_2) \backslash (R_1 \cap O_2) \backslash (R_2 \cap O_1)$.

Examples of component specifications and composition can be found in [31]. Client-server systems are often seen as applications in component software. The architecture of such a system is organized as a layered structure and can be model with in our model as shown in the full version [30] of paper [31].

6 Conclusion

We have proposed a classical relational model (rCOS) for component-based and object-oriented development. This model provides a smooth link between component-based design and object-oriented development. It supports rigorous application of UML in an iterative and incremental development process (RUP). The formalism is based on the design calculus in Hoare and He's Unifying Theories of Programming [19]. In a top-down process, model provides the fundamental basis for Model Driven Development. If we take a bottom-up approach, it supports re-engineering. Our message is: in order to support programming in the large,

- we need a multi-view modelling approach,
- a multi-notational modelling language is of a great advantage (though not everyone has to use UML),
- consistent refinement of different views is important,
- different verification techniques may be applied to refinement of different views.

The semantic model allows us to specify a system at different levels of abstraction. At the requirement, we can specify the functional requirements as use-case operations defined as methods of use-case controller classes. Each of these operations can be abstractly specified as a design in terms object creation, object destruction, and object attributes modification. Objects at these levels can be decomposed latter. These use-case operations can then refinement using the refinement laws for expert pattern, low coupling, high cohesion and attribute encapsulation. For details of requirement analysis and design by refinement, we refer the reader to [26,32]. With the refinement laws, algebraic reasoning is also supported.

This paper not only presents a semantics, but also provides a definitional approach to defining different semantics with different constraints and features.

6.1 Related Work

In the framework of ROOL [8], Borba, *et al*, also investigate refinement of object systems in [5]. Although, ROOL and rCOS share a number of common refinement laws, rCOS supports more features, such as references, and enjoys more refinement laws than ROOL.

There is an increasing amount of research in formal techniques for component-based development, e.g. [7,3,14]. These models are channel-based and process oriented. They can easily related to state machine models or automata and thus existing verification techniques and model checking tools can be readily applied. These models are flexible in describing interactions and coordinations among components due to the fine granularity of the interaction actions, that is channel-based message passing. We are aiming at a definitional semantic model that is easier to be related to software engineering concepts and terminology, such as *provided services*, *required services* and *contracts*, and programming languages, object-oriented languages in particular. We hope that this model will provide effective formal support to model-based development by pattern-guided transformations.

rCOS is motivated by our work on formal support to UML-based software development. We have studied the application of rCOS to support UML-based requirement modelling, analysis and design process. rCOS is used in [32] for formalisation of UML models of requirements, but a requirement model there only consists of a conceptual class diagram and a use-case model directly specified by rCOS. Article [28] uses rCOS for the specification of design class diagrams and sequence diagrams, but without rules for model transformation. A tool for requirement analysis has been developed using this framework [27]. Algorithms are also designed for consistency checking and executable code generation from a system model [35]. The technical report [16] presents detailed study of rCOS for object systems with examples and proofs of laws. A case study of the use of the refinement laws in software development is given in [36].

These publications show how rCOS can be used to support engineering methods and processes in software development.

6.2 Future Work

Future work includes the completion of the calculus rCOS for synchronization and concurrency in both object and component systems. This requires the model of components to be extended with the specification of the protocol in the interface, contract of a component. We plan to use traces (or regular expressions of operations for this purpose. The notion of designs of operations of *active* and *reactive* components have to be extended to *reactive designs* to capture synchronization. Consistency between the protocol and the reactive designs have to be checked to avoid from deadlock and divergence.

We will work on case studies to test the theory and the method. We are also interested in a theory of tool integration within this framework.

Acknowledgments

We are grateful to the organizers of FMCO'04 to invite Zhiming Liu to give a talk at the simposium and to the participants of the simposium for their comments. We would like thank Dines Bjorner at Technical University of Denmark, Kung-Kiu Lau at Manchester University, Anders Ravn at Aalborg University of Denmark and Uday Reddy from Birmingham University of the UK for their helpful comments and discussions at and after the seminars that Zhiming Liu gave on parts of the works when he visited them. Our UNU-IIST fellows Jing Liu, Xiaojian Liu, Quan Long, Bhim Upadhyaya and Jing Yang contributed to the whole research project. They also read and gave useful comments on this article. Zhiming Liu would also like to thank the students at the University of Leicester and those participants of the UNU-IIST training schools and courses who took his courses on Software Engineering and System Development with UML for their feedback on the understanding of the use-case driven, incremental and iterative object-oriented development and design patterns. Part of the work was also presented at the Workshop on Predictable Software Component Assembly held in the University of Manchester in 2004, and as an invited talk in Brazilian Symposium on Formal Methods (SBMF 2004). The discussion and comments were very much useful in bringing the presentation of the work into its current form.

References

1. M. Abadi and L. Cardeli. *A Theory of Objects*. Springer-Verlag, 1996.
2. R. Allen and D Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3), 1997.
3. F. Arbab. Reo: A channel-based coordination of model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
4. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.
5. P. Borba, A. Sampaio, and M. Cornélio. A refinment algebra for object-oriented programming. In L. Cardelli, editor, *Proc. ECOOP03, LNCS2743*, pages 457–482. Springer, 2003.

6. M. Broy. Object-oriented programming and software development - a critical assessment. In A. McIver and C. Morgan, editors, *Programming Methodology*. Springer, 2003.
7. M. Broy and K. Stolen. *Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement*. Springer, 2001.
8. A. Cavalcanti and D.A. Naumann. A weakest precondition semantics for an object-oriented language of refinement. Technical Report CS Report 9903, Stevens Institute of Technology, Hoboken, NJ 07030, February 2000.
9. J. Cheesman and J. Daniels. *UML Components. Component Software Series*. Addison-Wesley, 2001.
10. Y. Chen and J.W. Sanders. The weakest specifunction. *Acta Informatica*, 41(7), 2005.
11. J.K. Filipe. A logic-based formalization for component specification. *Journal of Object Technology*, 1(3):231–248, 2002.
12. M. Fowler. What is the point of UML. In P. Srevens, J. Whittle, and G. Booch, editors, *<<UML>> 2003 -The Unified Modeling Language, 6th International Conference, LNCS 2863*, San Fancisco, CA, USA, 2003. Springer.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addlison Wesley, 1995.
14. G. Goessler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*.
15. J.A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.
16. J. He, Z Liu, and X. Li. rCOS: A refinement calculus for object systems. Technical Report 322, UNU/IIST, P.O. Box 3058, Macao SAR China, 2005.
17. J. He, Z. Liu, X. Li, and S. Qin. A relational model of object oriented programs. In *Proceedings of the Second ASIAN Symposium on Programming Languages and Systems (APLAS04), LNCS 3302*, pages 415–436, Taiwan, March 2004. Springer.
18. G.T. Heineman and W.T. Councill. *Component-Based Software Engineering, Putting the Pieces Together*. Addison-Wesley, 2001.
19. C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
20. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
21. N. Jin and J. He. Resource models and pre-compiler specification for hardware/software. In J.R. Cuellar and Z. Liu, editors, *Proc. 2nd International Conference on Software Engineering and Formal Methods (SEFM'04)*, Beijing, China, 28-30 September, 2004. IEEE Computer Society.
22. C.B. Jones. Process algebra arguments about an object-oriented design notation. In A.W Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*. Prentice-Hall, 1994.
23. C.B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, 1996.
24. P. Kruchten. *The Rational Unified Process – An Introduction (2nd Edition)*. Addison-Wesly, 2000.
25. C. Larman. *Applying UML and Patterns*. Prentice-Hall International, 2001.
26. X. Li, Z. Liu, and J. He. Formal and use-case driven requirement analysis in UML. In *COMPSAC01*, pages 215–224, Illinois, USA, October 2001. IEEE Computer Society.
27. X. Li, Z. Liu, J. He, and Q. Long. Generating prototypes from a UML model of requirements. In *International Conference on Distributed Computing and Internet Technology(ICDIT2004), LNCS 3347*, pages 255–265, Bhubaneswar, India, 2004. Springer.
28. J. Liu, Z. Liu, J. He, and X. Li. Linking UML models of design and requirement. In *Proceedings of ASWEC2004*, pages 329–338, Melbourne, Australia, 2004. IEEE Computer Society.
29. Z. Liu. Object-oriented software development in UML. Technical Report UNU/IIST Report No. 228, UNU/IIST, P.O. Box 3058, Macau, SAR, P.R. China, March 2001.

30. Z. Liu, J. He, and X. Li. Contract-oriented component software development. Technical Report UNU/IIST, Report No 298, 2004. http://www.iist.unu.edu/newrh/III/1/page.html.
31. Z. Liu, J. He, and X. Li. Contract-oriented development of component systems. In *Proceedings of IFIP WCC-TCS2004*, pages 349–366, Toulouse, France, 2004. Kulwer Academic Publishers.
32. Z. Liu, J. He, X. Li, and Y. Chen. A relational model for formal requirements analysis in UML. In J.S. Dong and J. Woodcock, editors, *Formal Methods and Software Engineering, ICFEM03, LNCS 2885*, pages 641–664. Springer, 2003.
33. Z. Liu, J. He, X. Li, and J. Liu. Unifying views of UML. *Electronic Notes of Theoretical Computer Science (ENTCS)*, 101:95–127, 2004.
34. Q. Long, J. He, and Z. Liu. Refactoring and pattern-directed refactoring: A formal perspective. Technical Report 318, UNU-IIST, P.O.Box 3058, Macau, January 2005.
35. Q. Long, Z. Liu, X. Li, and J. He. Consistent code generation from UML models. In *Pro. of Australian Software Engineering Conference (ASWEC'2005)*, pages 168–177, Brisbane, Australia, 2005. IEEE Computer Sciety.
36. Q. Long, Z. Qiu, Z. Liu, L. Shao, and J. He. POST: A case study for rcos incremental development. Technical Report 324, UNU/IIST, P.O. Box 3058, Macao SAR China, 2005.
37. S.J. Mellor and M.J. Balcer. *Executable UML: a foundation for model-driven architecture*. Addison-Wesley, 2002.
38. B. Meyer. From structured programming to object-oriented design: the road to Eiffel. *Structured Programming*, 10(1):19–39, 1989.
39. B. Meyer. Applying design by contract. *IEEE Computer*, May 1992.
40. B. Meyer. *Object-oriented Software Construction (2nd Edition)*. Prentice Hall PTR, 1997.
41. C. Pierik and F.S. de Boer. A syntax-directed hoare logic for object-oriented programming concepts. Technical Report UU-CS-2003-010, Institute of Information and Computing Science, Utrecht University, 2003.
42. B. Selic. Using UML for modelling complex real-time systems. In F. Muller and A. Bestavros, editors, *Languages, compilers, and Tools for Embedded Systems, Volume 1474 of Lecture Notes in Computer Science*, pages 250–262. Springer Verlag, 1998.
43. A. Sherif, J. He, A Cavalcanti, and A. Sampaio. A framework for specification and validation of real-time systems using circus actions. In Z. Liu and K. Araki, editors, *Theoretical Aspects of Computing ICTAC 2004. First International Colloquium Guiyang, China, September 2004, Revised Selected Papers. LNCS 3407*, pages 478 – 494. Springer, 2005.
44. M. Smyth. Powerdomain. *Journal of Computer Science and System Sciences*, 16:23–36, 1978.
45. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.
46. J.M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.
47. J. Yang, Q. Long, Z. Liu, and X. Li. A predicative semantic model for integrating UML models. In Z. Liu and K. Araki, editors, *Theoretical Aspects of Computing – ICTAC 2004, First International Colloquium, Guiyang, China, September 2004, Revised Selected Papers, LNCS 3407*, pages 170–186. Springer, 2005.

Program Generation and Components

D. Ancona and E. Moggi*

DISI, Univ. of Genova, v. Dodecaneso 35, 16146 Genova, Italy
{davide, moggi}@disi.unige.it

Abstract. The first part of the paper gives a brief overview of meta-programming, in particular program generation, and its use in software development. The second part introduces a basic calculus, related to FreshML, that supports program generation (as described through examples and a translation of MetaML into it) and programming in-the-large (this is demonstrated by a translation of CMS into it).

1 Introduction

This paper explains what is program generation and what are the most promising uses of it, recalls the role of names in software components, and then presents a basic calculus, called MML_ν^N, which in the authors' view is a suitable formalism to describe program generation in terms of more primitive notions, namely name generation, name resolution and linking.

In calculi of module systems as CMS [AZ99, MT00, WV00, AZ02] modules can refer to *deferred* components by means of names. These calculi provide primitive operators for linking modules and resolving external names of deferred components, thus supporting programming in-the-large [Car97]. Analogously, in the MML_ν^N calculus of [AM04] names are used to refer to external components which need to be provided from the outside (by a name resolver).

In module (and record) calculi names are taken from some infinite set. On the contrary, in MML_ν^N at any time during execution there is a finite set of names, which can be extended dynamically by a construct $\nu X.e$ for generating a *fresh* name, borrowed from FreshML of [SPG03].

Fraenkel and Mostowski's set theory (see [GP99]) provides the mathematical underpinning of name generation for FreshML and MML_ν^N, but to understand the operational semantics and type system there is no need to be acquainted with FM-sets. Besides names $X \in \mathsf{Name}$, the calculus has

- terms $e \in \mathsf{E}$, a closed term corresponds to an *executable* program;
- name resolvers, which denote partial functions $\mathsf{Name} \xrightarrow{fin} \mathsf{E}$ with finite domain. We write $r.X$ for the term obtained by applying r to *resolve* name X.

Terms include fragments $\mathsf{b}(r)e$, i.e. term e abstracted w.r.t. resolver r, which denote functions $(\mathsf{Name} \xrightarrow{fin} \mathsf{E}) \to \mathsf{E}$. We write $e\langle r \rangle$ for the term obtained by *linking* fragment e using resolver r.

* Supported by EU projects DART IST-2001-33477 and APPSEM-II IST-2001-38957.

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 222–250, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Remark 1. If resolvers were included in terms, we would get a λ-calculus with extensible records [CM94]; indeed, a record amounts to a partial function mapping names (of components) to their values. More precisely, $\mathsf{b}(r)e$ would become an abstraction $\lambda r.e$ and $e\langle r\rangle$ an application $e\ r$. Even when resolvers are second class terms, one can express in the calculus the staging constructs of MetaML [Tah99, She01] and the mixin operations of CMS.

The ability to generate a fresh name is essential to prevent accidental overriding. If we know in advance what names need to be resolved within a fragment (we call such a fragment closed), then we can statically choose a name which is fresh (for that fragment). However, generic functions manipulating *open* fragments will have to generate fresh names at run-time. There are several reasons for working with open fragments: reusability is increased, the need for naming conventions (between independent developers) is reduced, and decisions can be delayed.

We present MML_ν^N as a monadic metalanguage, i.e. its type system makes explicit which terms have computational effects, and its operational semantics is given by a (semantic preserving) simplification relation on terms and a computation relation on *configurations*. Generation of fresh names is a computational effect, as in FreshML, thus typing $\nu X.e$ requires computational types.

Summary. Section 2 explains program generation within the broader context of meta-programming, and mentions its most promising uses. Section 3 recalls the role of names in software components. Section 4 recalls syntax, type system and operational semantics of MML_ν^N. Section 5 gives several programming examples: programming with open fragments, and benchmark examples for comparison with other calculi (in particular MetaML). Section 6 introduces a 2-level version of MetaML, and show how it can be translated into MML_ν^N. Section 7 introduces ML_Σ^N, a variant of MML_ν^N with records and recursive definitions, where one can recover all (mixin) module operations of CMS. Finally, Section 8 discusses related calculi based on names, i.e. FreshML of [SPG03] and $\nu^\Box$ of [NPar].

Notation. In the paper we use the following notations and conventions.

- m ranges over the set $\mathcal{N}$ of natural numbers. Furthermore, $m \in \mathcal{N}$ is identified with the set $\{i \in \mathcal{N} | i < m\}$ of its predecessors.
- Term equivalence $\equiv$ is α-conversion. $\mathrm{FV}(e)$ is the set of variables free in e, while $e[x_i{:}\, e_i \mid i \in m]$ denotes parallel capture avoiding substitution.
- $f{:}\, A \xrightarrow{fin} B$ means that f is a partial function from A to B with a finite domain, written $\mathsf{dom}(f)$. The image of f is denoted by $\mathsf{img}(f)$. $A \to B$ denotes the set of total functions from A to B. We use the following operations:
  - $\{a_i{:}\, b_i | i \in m\}$ is the partial function mapping a_i to b_i (where the a_i must be different, i.e. $a_i = a_j$ implies $i = j$);
    $\emptyset$ is the everywhere undefined partial function;
  - $f_{\setminus a}$ denotes the partial function f' s.t. $f'(a') = b$ iff $b = f(a')$ and $a' \neq a$;

  - $f\{a{:}\,b\}$ denotes the (partial) function f' s.t. $f'(a) = b$ and $f'(a') = f(a')$ when $a' \neq a$;
  - f, f' denotes the union of two partial functions with disjoint domains.
- $A\#B$ means that the sets A and B are disjoint.

2 Program Generation

We explain what is program generation by placing it in the broader context of meta-programming. We borrow from Sheard's invited talk at SAIG'01 [She01], which in addition discusses several areas of meta-programming research. Then we mention some of the most promising uses of program generation in the context of software development. We make no attempt to be exhaustive, instead we advise the interested reader to browse through the proceedings of the conference on Generative Programming and Component Engineering (GPCE) [BCT02, PS03, KV04], and a compendium [LBCO04] of contributions presented at a Dagstuhl seminar on "Domain-specific program generation", where a new IFIP WG on "Program Generation" was proposed.

2.1 What Is It?

In general programs manipulate data. Meta-programs are programs that manipulate object-programs, or more precisely data representing other programs. We are not committed to a particular (programming) language, thus by object-program we mean a syntactic element in a formal language. Broadly speaking we can classify meta-programs in three categories:

- Generators, which construct object-programs. For instance, specializers generate a specialized program solving an instance of a general problem.
- Analyzers, which analyze the structure of object-programs. For instance, type-checkers or tools that perform various kinds of static analysis.
- Transformers, which combine the features of analyzers and generators. For instance, optimizers that perform source-to-source transformation, or aspect weavers that insert code to address a cross-cutting concern.

A compiler is a typical example of program where the three kinds of meta-programs co-exist: a static analyzer for the source language, an optimizer for some intermediate language, and a program generator for the target language.

Object-programs can be represented at different level of abstraction (we call *code* the data representing a program):

- White-box abstraction, where code is text (i.e. a string) or an abstract syntax tree (AST). This representation is the most versatile, since it gives a low-level description of the object-program, but could be error prone.
- White-box abstraction modulo α-conversion. This representation of syntax has been considered in the context of logical frameworks to deal with binders in object languages, alongside other representations like higher-order abstract syntax. FreshML [GP99, SPG03] is the leading programming language that supports this abstraction.

– Black-box abstraction, where code can be executed and combined, but not analyzed. This is the most abstract representation.

The black-box abstraction is incompatible with program analyzers and transformers. On the other hand, program generators can work with any of the abstractions, and the black-box abstraction ensures the maximum separation of concerns between the generator and the user of the generated code.

We consider related concepts to clarify how they differ from the concept of code and meta-program. A program *configuration* is a snapshot of a program during execution, a *computation* is a description of the program execution, while code represents (the syntax of) a program independently from execution. A reflective program is a program that manipulates *itself*, thus it is a particular instance of a meta-program. One can identify three forms of reflection:

– Introspection is the ability of a program to analyze itself, namely its code (this introspection is called structural reflection) or its current configuration or execution history (this introspection is called behavioral reflection).
– Self-modification is the ability to modify itself.
– Intercession is the ability to manipulate its semantics, i.e. the interpreter or virtual machine for the reflective program.

A computation is *staged* when it is decomposed into stages along the temporal dimension. The change of stage is usually triggered by the acquisition of new information. In a meta-programming system supporting program generation there are two natural stages: the computation of the meta-program and the computation of the generated object-program. Depending on the nature of the meta-program, its computation is called differently (e.g. generation-time, compile-time, design-time, specialization-time), while the computation of the object-program is usually called run-time or use-time computation. Moreover, if the meta-programming system is *homogeneous*, i.e. the object-language for describing object-programs coincides with the meta-language, then it provides a natural support for *multi-stage* programming. MetaML [TS97] and MetaOCaml [CTHL03, Met01] are among the leading multi-stage programming languages. For several applications *heterogeneous* meta-programming systems are enough. In this case, the main issue is to provide support for a variety of object-languages.

2.2 What Is for?

Generative and component approaches have the potential to revolutionize software development in a similar way as automation and components revolutionized manufacturing. Generative programming (developing programs that synthesize other programs), component engineering (raising the level of modularization and analysis in application design), and domain-specific languages (elevating program specifications to compact domain-specific notations that are easier to write and maintain) are key technologies for automating program development. Before focusing on program generation, we mention some trends in software engineering, in order to provide a broader picture. [GS04] identifies *software factories* as the next methodology for software development. A software factory is

> a collection of *reusable* assets (like patterns, models, frameworks, tools) for *rapidly* and *cheaply* producing an *open-ended* set of unique variants of a software *product*.

Clearly, for a software product that has a big market and need to evolve, like an operating system, this is an economically feasible approach. However, to make software factories economically feasible for specific domains, it is essential to empower the domain-experts and end-users.

Domain-specific languages (DSLs) are a way to give programming abilities to domain-experts and end-users. Descriptions given in a DSL can be treated as high-level source code, rather than non-executable requirement specifications. Relational databases are a "classic" example of success story in the use of DSLs, while UML is a counter-example of DSL. In fact, UML is too general (i.e. it is not meant for any specific domain) and too imprecise (e.g. it cannot be used as source code, although some subsets might). Other examples of domains where DSL technology has been used successfully or appear highly promising are: language parsers, reactive real-time programs and the telephony domain. For instance, MetaCase Consulting (`http://www.metacase.com`) gave a demo at GPCE'03 entitled "MetaEdit+ revolutionized the way Nokia develops mobile phone software", and the choice of MetaEdit+ was motivated as follows

> When Nokia was searching for an effective CASE tool, the prime criteria was encapsulation of domain knowledge, flexible method support and code generation. After evaluating a number of off-the-shelf CASE tools, they undertook the development of their own solution using the MetaEdit+ metaCASE tool.

DSLs provide language-based abstraction, which goes well beyond the abstraction provided by libraries (i.e. platform extension). In this context program generators play the role of compiler back-ends for domain-specific languages, and provide the following benefits

- Increase automation by exploiting domain features and knowledge.
- Improve performance via partial evaluation.

Usually program generators are *co-designed* with the DSL they implement, unlike compilers for general-purpose languages. It is worth to adopt this approach, when the effort to implement a program generator is comparable to that of implementing a software library for the specific domain. In the DSL approach there are other things one can do before the program generation stage, namely

- analysis, which exploits domain-knowledge to identify problems prior generation, or to provide static guarantees
- transformation, for instance to perform domain-specific optimization.

It is important that analysis and transformation take place before program generation, so that they can be understood and managed by the user of the DSL, who can be expected to be knowledgeable of the domain but not of the target language for the program generator.

3 Names and Software Components

It has been argued that the notion of software component is so general that cannot be defined in a precise and comprehensive way [CE00]. For instance, [Szy02] provides three different definitions, that adopt different levels of abstraction. Nevertheless, names play an important role, independently of the particular definition adopted for software components. For clarity, we identify the notion of software component with that of *mixin module*[1], as done in CMS of [AZ02].

A basic module could be described as follows

import X_1 as x_1, ..., X_m as x_m
export $Y_1 = E_1$, ..., $Y_n = E_n$
local $z_1 = E'_1$, ..., $z_p = E'_p$

A basic module make use of *names* and *variables*. The former are the names of the components the module either *imports* from the outside (*input* components X_1, ..., X_m) or *exports* to the outside (*output* components Y_1,..., Y_n). The latter are the variables used in definitions inside the module (i.e. in the expressions E_1,..., E_n, E'_1,..., E'_p). These variables can be either *deferred* (x_1,..., x_n), i.e. associated with some input component, or locally defined (z_1,..., z_p).

The distinction between names and variables is crucial: names correspond to external references, while variables correspond to internal references. Technically speaking the main differences between variables and names are: expressions include variables but not names; variables declared in a basic module are local and can be α-converted; while component names belong to a global name space and allow modules to talk to each other.

A useful operator which can be easily encoded in CMS is the *link* operator $link(M_1, M_2)$, used for merging two modules and resolving input names. This operator may be regarded as either an operation provided by a module language in order to define structured module expressions or an extra-linguistic mechanism to combine object files provided by a tool for modular software development. $link(M_1, M_2)$ is well-defined if the sets of the output components of M_1 and M_2 are disjoint. In this case, $link(M_1, M_2)$ corresponds to a module where some input component of one module has been bound to the definition of the corresponding output component of the other module, and conversely.

For instance, let the modules `BOOL` and `INT` define the evaluation of some boolean and integer expressions in a mutually recursive way:

```
module BOOL is
  import IntEv as ext_ev
  export BoolEv = ev
  local
   fun ev EQ(ie1,ie2) = ext_ev(ie1)==ext_ev(ie2)
       |  ...
end BOOL;
```

[1] In the sequel we will interchangeably use the terms "module" and "mixin" as abbreviations of "mixin module".

```
module INT is
  import BoolEv as ext_ev
  export IntEv = ev
  local
   fun ev IF(be,ie1,ie2) =  if ext_ev(be) then ev(ie1) else ev(ie2)
       |  ...
end INT;
```

The result of *link*(BOOL,INT) corresponds to the module

```
module BOOL_INT is
  export IntEv = iev
  export BoolEv = bev
  local
   fun bev EQ(ie1,ie2) = iev(ie1)==iev(ie2)
       |  ...
   fun iev IF(be,ie1,ie2) =  ifbev(be) then iev(ie1) else iev(ie2)
       |  ...
end BOOL_INT;
```

The separation between component names and variables allows one to use internally the same name ev for the evaluation function in the two modules; in the compound module, indeed, ev of BOOL and ev of INT are α-renamed to bev and iev, respectively.

The *link* operation described above can be decomposed in two steps. First, put together the declarations of the two arguments in one module, yielding

```
module
  import IntEv as ext_iev
  import BoolEv as ext_bev
  export IntEv = iev
  export BoolEv = bev
  local
   fun bev EQ(ie1,ie2) = ext_iev(ie1)==ext_iev(ie2)
       |  ...
   fun iev IF(be,ie1,ie2) =  if ext_bev(be) then iev(ie1) else iev(ie2)
       |  ...
end;
```

Then, bind import components with export components with the same name, yielding BOOL_INT. Formally, this corresponds to the fact that *link* is a derived operator which can be expressed by the *sum* and *freeze* basic operators of CMS.

CMS provides also a primitive operation for deleting module components, which allows redefinition of components when used in conjunction with the *link* operator. This is an important feature for enabling reuse of software components and amortizing the investment over multiple applications [Szy02].

4 A Core Calculus with Names: MML_{ν}^{N}

This section recalls the monadic metalanguage MML_{ν}^{N} of [AM04]. For simplicity, we focus on the key feature, i.e. *names*, and exclude imperative computations

and functional types. Moreover, we restrict the formal treatment to a simply typed language (Section 4.1), and recall only the main statements concerning type safety (Section 4.4). We refer to [AM04] for details and a polymorphic extension of the type system, which is essential for typing the examples on open fragments generators (see Example 1 in Section 5). The operational semantics is given according to the general pattern proposed in [MF03], namely by a confluent *simplification* relation $\longrightarrow$ defined as the *compatible closure* of a set of rewrite rules (see Section 4.2), and a *computation* relation $\longmapsto$ describing how *configurations* may evolve (see Section 4.3).

Names X are syntactically pervasive, i.e. they occur both in types and in terms. The term $\nu X.e$ allows to generate a *fresh* name for private use within e. Following FreshML of [SPG03], we consider generation of a fresh name a computational effect, therefore for typing $\nu X.e$ we need computational types.

We parameterize typing judgments w.r.t. a finite set of names, namely those that can occur (free) in the judgment. The mathematical underpinning for names is provided by [GP99]. In particular, properties are invariant w.r.t. name permutation (equivariance), but not w.r.t. name substitution.

The syntax of MML_ν^N is abstracted over symbolic names $X \in \mathsf{Name}$, basic types b, term variables $x \in \mathsf{X}$ and resolver variables $r \in \mathsf{R}$. The syntactic category of types and signatures (i.e. the types of resolvers) is parameterized w.r.t. a finite set $\mathcal{X} \subseteq_{fin} \mathsf{Name}$ of names that can occur in the types and signatures.

- $\boxed{\tau \in \mathsf{T}_\mathcal{X} ::= b \mid [\Sigma|\tau] \mid M\tau}$ $\mathcal{X}$-types, where
  $\Sigma \in \mathsf{\Sigma}_\mathcal{X} \stackrel{\Delta}{=} \mathcal{X} \stackrel{fin}{\rightarrow} \mathsf{T}_\mathcal{X}$ is a $\mathcal{X}$-signature $\{X_i{:}\,\tau_i | i \in m\}$
- $\boxed{e \in \mathsf{E} ::= x \mid \theta.X \mid e\langle\theta\rangle \mid \mathsf{b}(r)e \mid \mathsf{ret}\ e \mid \mathsf{do}\ x \leftarrow e_1; e_2 \mid \nu X.e}$ terms, where
  $\theta \in \mathsf{ER} ::= r \mid ? \mid \theta\{X{:}\,e\}$ is a name resolver term.

We give an informal semantics of the language (see Section 5 for examples).

- The type $[\Sigma|\tau]$ classifies fragments which produce a term of type τ when linked with a resolver for Σ. The terms $\theta.X$ and $e\langle\theta\rangle$ use θ to *resolve* name X and to *link* fragment e. The term $\mathsf{b}(r)e$ *represents* the fragment obtained by abstracting e w.r.t. r.
- The resolver ? cannot resolve any name, while $\theta\{X{:}\,e\}$ resolves X with e and *delegates* the resolution of other names to θ.
- The monadic type $M\tau$ classifies programs computing values of type τ. The terms $\mathsf{ret}\ e$ and $\mathsf{do}\ x \leftarrow e_1; e_2$ are used to terminate and sequence computations, $\nu X.e$ generates a *fresh* name for use within the computation e.

As a simple example, let us consider the fragment `b(r)(r.X*r.X)` which can be correctly linked with resolvers mapping `X` to integer expressions and whose type is `[X:int|int]`. Then we can link the fragment with the resolver `?{X:2}`, as in `b(r)(r.X*r.X)<?{X:2}>`, and obtain `2*2` of type `int`. Note that `b(r)(r.X*r.X)` is not equivalent to `b(r)(r.Y*r.Y)`, whose type is `[Y:int|int]`. This is in clear contrast with what happens with variables and λ-abstractions: `\x->x*x` and `\y->y*y` are equivalent and have the same type. The sequel of this

section is devoted to the formal definition of MML_{ν}^{N}. More interesting examples (with informal explanatory text) can be found in Section 5.

One can define (by induction on τ, e and θ) the following syntactic functions:

- the set $\mathrm{FV}(_) \subseteq_{fin} \mathsf{Name} \uplus \mathsf{X} \uplus \mathsf{R}$ of free names and variables in $_$, in particular $\mathrm{FV}(\{X_i{:}\,\tau_i | i \in m\}) = (\cup_{i \in m} \mathrm{FV}(\tau_i)) \cup \{X_i | i \in m\}$
- the capture-avoiding substitution $_[x_0{:}\,e_0]$ for term variable x_0.
- the capture-avoiding substitution $_[r_0{:}\,\theta_0]$ for resolver variable r_0.
- the action $_[\pi]$ of a name permutation π on $_$.

4.1 Type System

The typing judgments are $\mathcal{X}; \Pi; \Gamma \vdash e{:}\,\tau$ (i.e. e has type τ) and $\mathcal{X}; \Pi; \Gamma \vdash \theta{:}\,\Sigma$ (i.e. θ resolves the names in the domain of Σ, and only them, with terms of the assigned type), where

- τ is a $\mathcal{X}$-type and Σ is a $\mathcal{X}$-signature
- $\Pi{:}\,\mathsf{R} \xrightarrow{fin} \mathsf{\Sigma}_{\mathcal{X}}$ is a signature assignment $\{r_i{:}\,\Sigma_i | i \in m\}$ for resolver variables
- $\Gamma{:}\,\mathsf{X} \xrightarrow{fin} \mathsf{T}_{\mathcal{X}}$ is a type assignment $\{x_i{:}\,\tau_i | i \in m\}$ for term variables

The typing rules are given in Table 1. All the rules, except that for $\nu X.e$, use the same finite set $\mathcal{X}$ of names in the premises and the conclusion. The typing rule for $e\langle\theta\rangle$ supports a limited form of *width* subtyping, namely it allows linking of a fragment $e{:}\,[\Sigma|\tau]$ with a resolver θ whose signature Σ' includes Σ. All the other rules are standard.

4.2 Simplification

We define a confluent relation on terms, called *simplification*. There is no need to define a deterministic simplification strategy, since computational effects are *insensitive* to further simplification. Simplification $\longrightarrow$ is the compatible closure of the following rules

(resolve) $(\theta\{X{:}\,e\}).X \longrightarrow e$
(delegate) $(\theta\{X{:}\,e\}).X' \longrightarrow \theta.X'$ if $X' \neq X$
(link) $(\mathsf{b}(r)e)\langle\theta\rangle \longrightarrow e[r{:}\,\theta]$

Simplification enjoys the following properties.

Theorem 1 (Church-Rosser). *The simplification relation* $\longrightarrow$ *is confluent.*

Theorem 2 (Subject Reduction).

- *If* $\mathcal{X}; \Pi; \Gamma \vdash e{:}\,\tau$ *and* $e \longrightarrow e'$, *then* $\mathcal{X}; \Pi; \Gamma \vdash e'{:}\,\tau$.
- *If* $\mathcal{X}; \Pi; \Gamma \vdash \theta{:}\,\Sigma$ *and* $\theta \longrightarrow \theta'$, *then* $\mathcal{X}; \Pi; \Gamma \vdash \theta'{:}\,\Sigma$.

Table 1. Type System for MML_ν^N

$$x\ \frac{}{\mathcal{X};\Pi;\Gamma \vdash x{:}\tau}\ \Gamma(x)=\tau \qquad \text{resolve}\ \frac{\mathcal{X};\Pi;\Gamma \vdash \theta{:}\Sigma}{\mathcal{X};\Pi;\Gamma \vdash \theta.X{:}\tau}\ \tau=\Sigma(X)$$

$$\text{link}\ \frac{\begin{array}{c}\mathcal{X};\Pi;\Gamma \vdash e{:}[\Sigma|\tau] \\ \mathcal{X};\Pi;\Gamma \vdash \theta{:}\Sigma'\end{array}}{\mathcal{X};\Pi;\Gamma \vdash e\langle\theta\rangle{:}\tau}\ \Sigma\subseteq\Sigma' \qquad \text{box}\ \frac{\mathcal{X};\Pi,r{:}\Sigma;\Gamma \vdash e{:}\tau}{\mathcal{X};\Pi;\Gamma \vdash \mathsf{b}(r)e{:}[\Sigma|\tau]}$$

$$r\ \frac{\Pi(r)=\Sigma}{\mathcal{X};\Pi;\Gamma \vdash r{:}\Sigma}\qquad ?\ \frac{}{\mathcal{X};\Pi;\Gamma \vdash ?{:}\emptyset}\qquad \text{extr}\ \frac{\begin{array}{c}\mathcal{X};\Pi;\Gamma \vdash \theta{:}\Sigma \\ \mathcal{X};\Pi;\Gamma \vdash e{:}\tau\end{array}}{\mathcal{X};\Pi;\Gamma \vdash \theta\{X{:}e\}{:}\Sigma\{X{:}\tau\}}$$

$$\text{ret}\ \frac{\mathcal{X};\Pi;\Gamma \vdash e{:}\tau}{\mathcal{X};\Pi;\Gamma \vdash \mathsf{ret}\ e{:}M\tau} \qquad \text{do}\ \frac{\begin{array}{c}\mathcal{X};\Pi;\Gamma \vdash e_1{:}M\tau_1 \\ \mathcal{X};\Pi;\Gamma,x{:}\tau_1 \vdash e_2{:}M\tau_2\end{array}}{\mathcal{X};\Pi;\Gamma \vdash \mathsf{do}\ x \leftarrow e_1;e_2{:}M\tau_2}$$

$$\nu\ \frac{\mathcal{X},X;\Pi;\Gamma \vdash e{:}M\tau}{\mathcal{X};\Pi;\Gamma \vdash \nu X.e{:}M\tau}\ X\notin \mathrm{FV}(\Pi,\Gamma,\tau)$$

4.3 Computation

The computation relation $Id \longmapsto Id' \mid \mathsf{done}$ is defined using evaluation contexts and configurations $Id \in \mathsf{Conf}$. A configuration records the current name space as a finite set $\mathcal{X}$ of names. The computation rules (see Table 2) consist of those given in [MF03] for the monadic metalanguage MML (these rules do not change the name space) plus a rule for generation of a fresh name (this is the only rule that extends the name space).

- $\boxed{E \in \mathsf{EC} ::= \Box \mid E[\mathsf{do}\ x \leftarrow \Box; e]}$ evaluation contexts
- $(\mathcal{X}|e,E) \in \mathsf{Conf} \stackrel{\Delta}{=} \mathcal{P}_{fin}(\mathsf{Name}) \times \mathsf{E} \times \mathsf{EC}$ configurations consist of the current name space $\mathcal{X}$ (which grows as computation progresses), the program fragment e under consideration, and its evaluation context E
- $\boxed{rc \in \mathsf{RC} ::= \mathsf{ret}\ e \mid \mathsf{do}\ x \leftarrow e_1; e_2 \mid \nu X.e}$ computational redexes.

Simplification $\longrightarrow$ is extended in the obvious way to a confluent relation on configurations (and related notions). The bisimulation property, i.e. computation is insensitive to further simplification, is like that stated in [MF03] for MML.

Theorem 3 (Bisimumation). *If* $Id \equiv (\mathcal{X}|e,E)$ *with* $e \in \mathsf{RC}$ *and* $Id \xrightarrow{*} Id'$, *then*

1. $Id \longmapsto D$ *implies* $\exists D'$ *s.t.* $Id' \longmapsto D'$ *and* $D \xrightarrow{*} D'$
2. $Id' \longmapsto D'$ *implies* $\exists D$ *s.t.* $Id \longmapsto D$ *and* $D \xrightarrow{*} D'$

where D *and* D' *range over* $\mathsf{Conf} \cup \{\mathsf{done}\}$.

Table 2. Computation Relation

Administrative steps

(A.0) $(\mathcal{X}|\mathsf{ret}\ e, \Box) \longmapsto \mathsf{done}$
(A.1) $(\mathcal{X}|\mathsf{do}\ x \leftarrow e_1; e_2, E) \longmapsto (\mathcal{X}|e_1, E[\mathsf{do}\ x \leftarrow \Box; e_2])$
(A.2) $(\mathcal{X}|\mathsf{ret}\ e_1, E[\mathsf{do}\ x \leftarrow \Box; e_2]) \longmapsto (\mathcal{X}|e_2[x{:}\, e_1], E)$

Name generation step

(ν) $(\mathcal{X}|\nu X.e, E) \longmapsto (\mathcal{X}, X|e, E)$ with X renamed to avoid clashes, i.e. $X \notin \mathcal{X}$

Table 3. Well-formed Evaluation Contexts

$$\Box \quad \frac{}{\mathcal{X}; \Box{:}\, M\tau \vdash \Box{:}\, M\tau} \qquad \frac{\mathcal{X}; \Box{:}\, M\tau_2 \vdash E{:}\, M\tau' \quad \mathcal{X}; \emptyset; x{:}\, \tau_1 \vdash e{:}\, M\tau_2}{\mathcal{X}; \Box{:}\, M\tau_1 \vdash E[\mathsf{do}\ x \leftarrow \Box; e]{:}\, M\tau'}$$

4.4 Type Safety

Following Felleisen, type safety can be decomposed in two properties: subject reduction and progress. We refer to [MF03] for a formulation of these properties in the context of a monadic metalanguage.

Definition 1 (Well-formed configuration). $\vdash (\mathcal{X}|e, E){:}\, \tau' \stackrel{\Delta}{\Longleftrightarrow} \tau' \in \mathsf{T}_\emptyset$ *and* $\exists \tau \in \mathsf{T}_\mathcal{X}$ *s.t.* $\mathcal{X}; \emptyset; \emptyset \vdash e{:}\, M\tau$ *and* $\mathcal{X}; \Box{:}\, M\tau \vdash E{:}\, M\tau'$ *(see Table 3).*

Theorem 4 (Subject Reduction).

- *If* $\vdash Id_1{:}\, \tau'$ *and* $Id_1 \longrightarrow Id_2$, *then* $\vdash Id_2{:}\, \tau'$.
- *If* $\vdash Id_1{:}\, \tau'$ *and* $Id_1 \longmapsto Id_2$, *then* $\vdash Id_2{:}\, \tau'$.

Theorem 5 (Progress). *If* $\vdash (\mathcal{X}|e, E){:}\, \tau'$, *then*

1. *either* $e \notin \mathsf{RC}$ *and* $e \longrightarrow$
2. *or* $e \in \mathsf{RC}$ *and* $(\mathcal{X}|e, E) \longmapsto$

5 Programming Examples

We demonstrate the use and expressivity of MML_ν^N with a few examples:

- the first exemplifies programming with *open* fragments;
- the second recasts the multi-stage programming method of [TS97] by making a simplified use of *open* fragments;

– the third uses *closed* fragments, and allows a further comparison with other calculi for run-time code generation and staging.

To improve readability we use ML-like notation for functions (β-reduction is a sound simplification in monadic metalanguages) and operations on references, and Haskell's do-notation `do {x1 <- e1; ...; xn <- en; e}`. In the sequence of commands of a do-expression we allow computations `ei` whose value is not bound to a variable (because it is not used by other commands) and non-recursive let-bindings like `xi = ei` (which amounts to replace `xi` with `ei` in the commands following the let-binding).

Example 1. We consider an example of generative programming, which motivates the need for fresh name generation. In our calculus, a component is identified with a fragment of type $[\Sigma|\tau]$, where Σ specifies what information needs to be provided for deployment. Generative programming supports dynamic manufacturing of customized components from elementary (highly reusable) components. The most appropriate building block for generative programming are polymorphic functions $G{:}\forall p.[p, \Sigma_i|\tau_i] \rightarrow M[p, \Sigma|\tau]$ (we refer to [AM04] for a polymorphic extension of the type system). The result type of G is computational, because generation may require computational activities, while the *signature variable* p classifies the information passed to the parameters of G, but not directly used or provided in the implementation of G itself. Applications of G may instantiate p with different signatures, thus we say that G manipulates *open* fragments. An over-simplified example of open fragment generator is

```
Ac: [p|a->a] -> M[p|{add: a -> M unit, update: M unit}]
```

`Ac` creates a data structure to maintain an (initially empty) set of accounts. Since we don't really need to know the structure of an account, we use a type variable `a`. The generator makes available two functionalities for operating on a set (of accounts): `add` inserts a new account in the set, and `update` modifies all the accounts in the set by applying a function of type `a->a`, which depends on certain parameters (e.g. the interest rate) represented by the signature variable `p`. These parameters are decided by the bank after the data structure has been created, and they change over time.

In many countries bank accounts are taxed, according to local criteria. So we need a more refined generator, with an extra parameter for computing the new balance based on the state of the account after the bank's update

```
TaxedAc: [p'|a->a] -> [p|a->a] ->
          M[p'|[p|{add: a -> M unit, update: M unit}]]
```

The signature variable `p'` classifies the information needed to compute local taxes. In general `p'` and `p` are unrelated, and identifying them means that banks and local authorities rely on the same information. `TaxedAc` is defined as follows

```
fun TaxedAc tax upd = nu Tax.
    do {m <- Ac(b(r2) fn x => r2.Tax (upd<r2> x));
        ret (b(r') b(r1) m<r1{Tax:tax<r'>}>)};
```

It is essential that the name `Tax` is fresh and private to `TaxedAc`, otherwise we may override some information in `r1`, which is needed by `upd`. In fact, `TaxedAc` is an open fragment generator that does not know in advance how the signature variable `p` could be instantiated. On the other hand, with *closed* fragment generators $G\colon [\Sigma_i|\tau_i] \to M[\Sigma|\tau]$ the problem does not arise, but reusability is impaired. For instance, it is not reasonable to expect that all banks will use the same parameters to update the accounts of their customers.

Example 2. We recast the *multi-stage programming method* of [TS97] (see also [CMS03]) using the power function, which is a classical example for staged programming.

1. The method starts from a "conventional" program `exp` with two parameters. In the specific example, `exp` takes an exponent n, a base x, and then computes x^n by making recursive calls

   ```
   fun exp n x = if n=0 then ret(1.0)
           else do {y <- (exp (n-1) x); ret(x*y)};
   >   exp = ... : int -> real -> M real
   ```

   The result type of `exp` is computational, because we consider recursion a computational effect.
2. Then one obtains a "staged" version `exp_a`, which replaces the second parameter (the base x) with an open fragment u, and builds an open fragment representing the desired result (i.e. x^n)

   ```
   fun exp_a n u = if n=0 then ret(b(r) 1.0)
             else do {v <- exp_a (n-1) u; ret(b(r) u<r>*v<r>)};
   >   exp_a = ... : int -> [p|real] -> M[p|real]
   ```

   The staged version is polymorphic in the signature variable `p`.
3. By exploiting the polymorphism of `exp_a`, one defines a *code generator* `exp_cg`. Given the base n, the generator calls `exp_a` with a "dummy" parameter `b(r) r.X`, then builds an open fragment representing a function

   ```
   fun exp_cg n = nu X. do {v <- exp_a n (b(r) r.X);
                            ret (b(r) fn x => v<r{X:x}>)};
   >   exp_cg = ... : int -> M[p|real -> real]
   ```

   The type of `exp_cg` says that recursion is unfolded at "specialization" time, when the exponent n is known.
4. By instantiating `p` with the empty signature, one gets an *optimized* program

   ```
   fun exp_o n = do {v <- exp_cg n; ret(v<?>)};
   >   exp_o = ... : int -> M(real -> real)
   ```

   The type of `exp_o` differs from the type of the conventional program `exp` to reflect the different timing in unfolding recursion. Namely, `exp_o` unfolds the recursion when the parameter n is known. For instance, when $n = 2$

```
   do  sq_o <- exp_o 2;
   >   sq_o = (fn x => x*(x*1.0)) : real -> real
```

The multi-stage programming method makes use of open fragments of type $[p|\tau]$ (these types are similar to the code types annotated with environment classifiers $\langle\tau\rangle^{\alpha}$ used by [TN03, CMT04]). One can easily recast the multi-stage programming method also in the presence of more complex computational effects (while in MetaML there are typing problems). For instance, when the conventional program is an imperative variant `p:int->real->(real ref)->M unit` of `exp`

```
fun p n x y = if n=0 then y:=1.0
      else do {p (n-1) x y; y' <- !y; y:=x*y'};
>   p = ... : int -> real -> (R real) -> M unit
```

`p` takes an exponent n, a base x and a reference `y`, then it initializes y with 1.0 and repeatedly multiplies the content of y with x until it becomes x^n. The "staged" version, `p_a`, is defined in the obvious way, and its type says that some computations are postponed to the second stage

```
fun p_a n u v= if n=0 then ret(b(r) v<r>:=1.0)
        else do {
             w <- p_a (n-1) u v;
             ret(b(r) do {w<r>; y' <-!v<r>; v<r>:=u<r>*y'})};
>   p_a = ... : int ->  [p|real] -> [p|Ref real] -> M[p|M unit]
```

In comparison to MetaML, we don't face the problems due to execution of *potentially open* code or *scope extrusion*, which motivated the introduction of closed types in [CMS03]. The reason is that in MML_{ν}^{N} one has a better control of the name space and name resolution.

Example 3. We reconsider the power function `exp:int->real->M real`, and give an alternative way to define `exp_o:int-> M(real-> real)`, which does not involve fresh name generation.

```
(* conventional program *)
fun exp n x = if n=0 then ret(1.0)
        else do {y <- (exp (n-1) x); ret(x*y)};
>   exp = ... : int -> real -> M real
(* staged program *)
fun exp_a n u = if n=0 then ret(b(r) 1.0)
          else do {v <- exp_a (n-1) u; ret(b(r) u<r>*v<r>)};
>   exp_a = ... : int -> [p|real] -> M[p|real]
(* exp_c generates a fragment with hook X for base *)
fun exp_c n = do {v <- exp_a n (b(r) r.X);
                  ret (b(r) fn x => v<r>)};
>   exp_c = ... : int -> M[X:real|real]
(* optimized program *)
fun exp_o n = do {v <- exp_c n; ret(fn x => u<?{X:x}>)};
>   exp_o = ... : int -> M(real -> real)
```

The definition of `exp_c` relies on a pre-existing name `X`, while `exp_cg` uses a freshly generated name. MetaML does not allow to mention names explicitly, thus it has no analogue of `exp_c` nor of the type `[X:real|real]`. On the other hand, $\nu^{\Box}$ has an analogue of `exp_c` (see `exp'` in [NPar, Example 2]), but the name `X` has to be declared of type `real` globally.

6 Relating MML_{ν}^{N} to MetaML

In this section we define a monadic CBV translation of a 2-level version of MetaML into MML_{ν}^{N} (extended with functional types), and show that the translation preserves the operational semantics. We make no formal claim about preservation of typing, since we have not been able to extend the translation to types. We have not defined a monadic CBV translation of the whole MetaML, since key ideas would get confused with orthogonal issues involved in the translation of a multi-level language. Restricting to a 2-level language allows to bring these ideas in the foreground.

6.1 MetaML_2

We give the formal definition (syntax, a simplified type system and big-step CBV operational semantics) of MetaML_2, a 2-level version of MetaML. As customary for 2-level languages, the syntax (type system and operational semantics) is stratified in two levels: the meta-level 0, and the object-level 1.

- $\tau^0 \in \mathsf{T}^0 ::= b \mid \tau_1^0 \to \tau_2^0 \mid \langle \tau^1 \rangle$
  $\tau^1 \in \mathsf{T}^1 ::= b \mid \tau_1^1 \to \tau_2^1$
  types at level 0 and 1, note that $\mathsf{T}^1 \subset \mathsf{T}^0$
- $e^0 \in \mathsf{E}^0 ::= \underline{v^0} \mid e_1^0 e_2^0 \mid \langle e^1 \rangle \mid \mathsf{run}\ e^0$
  $v^0 \in \mathsf{V}^0 ::= x^0 \mid \lambda x^0.e^0 \mid \langle v^1 \rangle$
  $e^1 \in \mathsf{E}^1 ::= \underline{v^1} \mid \lambda x^1.e^1 \mid e_1^1 e_2^1 \mid \tilde{}e^0$
  $v^1 \in \mathsf{V}^1 ::= x^1 \mid \lambda x^1.v^1 \mid v_1^1 v_2^1$
  terms and values at level 0 and 1

We give an informal semantics of the language.

- A value v^1 corresponds to an object-level program, while a term e^1 may require some meta-level computation to get an object-level program.
- The type $\langle \tau^1 \rangle$ classifies code, i.e. meta-level values of the form $\langle v^1 \rangle$ representing an object-level program v^1. Brackets $\langle e^1 \rangle$ and escape $\tilde{}e^0$ allow to move between meta-level code and the corresponding object-level program, in particular $\tilde{}\langle e^1 \rangle$ and e^1 evaluate to the same object-level program.
- The construct $\mathsf{run}\ e^0$ first evaluates e^0 to code $\langle v^1 \rangle$, then evaluates the object-level program v^1 (provided it is a *complete* program, i.e. $\mathrm{FV}(v^1) = \emptyset$).

Besides the stratification in two levels, there are the following syntactic differences between MetaML_2 and MetaML of [CMT04]:

- MetaML_2 values are explicitly marked in terms. This avoids re-evaluation and a general pitfall of monadic CBV translations, namely preservation of the operational semantics (as stated in Theorem 6) would fail.
- Cross-stage persistence, i.e. the ability to include meta-level values ($v^0 : \tau^1$) into object-level programs, is excluded from MetaML_2. This simplifies some definitions and technical lemmas.
- In MetaML_2 code types are not annotated with environment classifiers.

A type system (without the environment classifiers of [TN03, CMT04]) is given by the following rules, where n ranges over levels (i.e. is either 0 or 1):

$$x^n\ \frac{\Gamma(x^n) = \tau^n}{\Gamma \vdash_n x^n : \tau^n} \qquad \mathrm{val}\ \frac{\Gamma \vdash_n v^n : \tau^n}{\Gamma \vdash_n \underline{v^n} : \tau^n} \qquad \mathrm{brk}_v\ \frac{\Gamma \vdash_1 v^1 : \tau^1}{\Gamma \vdash_0 \langle v^1 \rangle : \langle \tau^1 \rangle}$$

$$\lambda^n\ \frac{\Gamma, x^n : \tau_1^n \vdash_n e^n : \tau_2^n}{\Gamma \vdash_n \lambda x^n.e^n : \tau_1^n \to \tau_2^n} \qquad @^n\ \frac{\Gamma \vdash_n e_1^n : \tau_1^n \to \tau_2^n \quad \Gamma \vdash_n e_2^n : \tau_1^n}{\Gamma \vdash_n e_1^n e_2^n : \tau_2^n}$$

$$\lambda_v\ \frac{\Gamma, x^1 : \tau_1^1 \vdash_1 v^1 : \tau_2^1}{\Gamma \vdash_1 \lambda x^1.v^1 : \tau_1^1 \to \tau_2^1} \qquad @_v\ \frac{\Gamma \vdash_1 v_1^1 : \tau_1^1 \to \tau_2^1 \quad \Gamma \vdash_1 v_2^1 : \tau_1^1}{\Gamma \vdash_1 v_1^1 v_2^1 : \tau_2^1}$$

$$\mathrm{run}\ \frac{\Gamma \vdash_0 e^0 : \langle \tau^1 \rangle}{\Gamma \vdash_0 \mathsf{run}\ e^0 : \tau^1} \qquad \mathrm{brk}\ \frac{\Gamma \vdash_1 e^1 : \tau^1}{\Gamma \vdash_0 \langle e^1 \rangle : \langle \tau^1 \rangle} \qquad \mathrm{esc}\ \frac{\Gamma \vdash_0 e^0 : \langle \tau^1 \rangle}{\Gamma \vdash_1 \tilde{e}^0 : \tau^1}$$

The operational semantics of Table 4 consists of two relations $e^n \xrightarrow{\subset}{}^n v^n$, that evaluate terms to values. The operational semantics uses only the substitution $_[x^0 : v^0]$ and enjoys the following properties.

Proposition 1 (Operational properties).

- demote $\dfrac{\{x_i^1 : \tau_i^1 | i \in m\} \vdash_1 v^1 : \tau^1}{\{x_i^0 : \tau_i^1 | i \in m\} \vdash_0 v^1 \downarrow : \tau^1}$
- SR $\dfrac{\Gamma \vdash_n e^n : \tau^n \quad e^n \xrightarrow{\subset}{}^n v^n}{\Gamma \vdash_n v^n : \tau^n}$

6.2 Translation of MetaML_2 into MML_ν^N

Table 5 defines (by induction on the syntax of MetaML_2) a translation $[\![e^n]\!]^\rho$, $[\![v^0]\!]^\rho$ and $[\![v^1]\!]^\rho_\theta$, where θ is a resolver and ρ is a (partial) mapping from variables of MetaML_2 to terms of MML_ν^N such that

- a meta-level variable x^0 is mapped to a term e
- an object-level variable x^1 is mapped to a fragment $\mathsf{b}(r)e$

The parameters ρ and θ are convenient to state some properties (see Lemma 1), but for the definition of the translation it suffices to take $\theta = r$ and $\rho(x^0) = x$.

Some clauses in the definition of the translation deserve to be commented:

- terms of the form $\underline{v^n}$ are always translated into terms of the form $\mathsf{ret}\ e$, since values v^n do not require meta-level computation

Table 4. Big-Step Operational Semantics for MetaML_2

$$\frac{}{\underline{v^n} \subset\!\!\xrightarrow{n} v^n} \qquad \frac{e^1 \subset\!\!\xrightarrow{1} v^1}{\langle e^1 \rangle \subset\!\!\xrightarrow{0} \langle v^1 \rangle} \qquad \frac{e^1 \subset\!\!\xrightarrow{1} v^1}{\lambda x^1.e^1 \subset\!\!\xrightarrow{1} \lambda x^1.v^1}$$

$$\frac{e_1^0 \subset\!\!\xrightarrow{0} \lambda x^0.e^0 \quad e_2^0 \subset\!\!\xrightarrow{0} v^0 \quad e^0[x^0\colon v^0] \subset\!\!\xrightarrow{0} v_1^0}{e_1^0 e_2^0 \subset\!\!\xrightarrow{0} v_1^0} \qquad \frac{e^0 \subset\!\!\xrightarrow{0} \langle v^1 \rangle \quad v^1 \downarrow \subset\!\!\xrightarrow{0} v^0}{\mathsf{run}\ e^0 \subset\!\!\xrightarrow{0} v^0}\ \mathrm{FV}(v^1) = \emptyset$$

$$\frac{e_1^1 \subset\!\!\xrightarrow{1} v_1^1 \quad e_2^1 \subset\!\!\xrightarrow{1} v_2^1}{e_1^1 e_2^1 \subset\!\!\xrightarrow{1} v_1^1 v_2^1} \qquad \frac{e^0 \subset\!\!\xrightarrow{0} \langle v^1 \rangle}{\tilde{}e^0 \subset\!\!\xrightarrow{1} v^1}$$

where demotion $v^1 \downarrow$ is defined by induction on v^1

$$x^1 \downarrow = \underline{x^0} \qquad (\lambda x^1.v^1) \downarrow = \underline{\lambda x^0.v^1 \downarrow} \qquad (v_1^1 v_2^1) \downarrow = v_1^1 \downarrow\ v_2^1 \downarrow$$

- the translations of $\langle e^1 \rangle$ and e^1 are the same (and similarly for $\tilde{}e^0$ and e^0), because the bijection between meta-level code and object-level programs is collapsed to an equality
- the translation of $\langle v^1 \rangle$ is a fragment, which results into an object-level program after linking, thus the translation of v^1 depends on a resolver θ
- the translations of $e_1^1 e_2^1$ and $\lambda x^1.e^1$ are meta-level computations to generate code representing an application and abstraction in the object language
- the translation of values at level 1 (i.e. object-level programs) is like the monadic CBV translation of the λ-calculus, as the object language is CBV.

There are problems in extending the translation to types (thus we make no formal claim about preservation of typing). More precisely, the problem is to identify a signature to replace of ... in the following inductive definition

$$[\![b]\!] = b \qquad [\![\tau_1^n \to \tau_2^n]\!] = [\![\tau_1^n]\!] \to M[\![\tau_2^n]\!] \qquad [\![\langle \tau^1 \rangle]\!] = [\ldots | M[\![\tau^1]\!]]$$

The translation preserves the operational semantics in the following sense:

Theorem 6. *$e^0 \subset\!\!\xrightarrow{0} v^0$ and $\mathrm{FV}(e^0) = \emptyset$ imply $(\emptyset | [\![e^0]\!], \Box) \stackrel{*}{\Longrightarrow} (\mathcal{X} | \mathsf{ret}\ [\![v^0]\!], \Box)$ for some $\mathcal{X}$, where $\Longrightarrow \stackrel{\Delta}{=} \longrightarrow \cup \longmapsto$.*

The result is a consequence of the following lemmas (stated without proof).

Lemma 1 (Properties of Translation).

1. *If $e = [\![v^0]\!]^\rho$, then $[\![_[x^0\colon v^0]]\!]^\rho = [\![_]\!]^{\rho, x^0 : e}$ and $[\![_[x^0\colon v^0]]\!]_\theta^\rho = [\![_]\!]_\theta^{\rho, x^0 : e}$*
2. *$[\![v^1]\!]_\theta^\rho = [\![v^1 \downarrow]\!]^{\rho'}$, if $\rho'(x^0) = e[r\colon\theta]$ when $\rho(x^1) = \mathsf{b}(r)e$ and $x^1 \in \mathrm{FV}(v^1)$*
3. *$[\![v^1]\!]_{\theta_1}^{\rho_1} \stackrel{*}{\longrightarrow} [\![v^1]\!]_{\theta_2}^{\rho_2}$, if $e_1[r\colon\theta_1] \stackrel{*}{\longrightarrow} e_2[r\colon\theta_2]$ when $\rho_i(x^1) = \mathsf{b}(r)e_i$ and $x^1 \in \mathrm{FV}(v^1)$.*

Table 5. Translation of MetaML_2 terms and values

e^0	$[\![e^0]\!]^\rho$
$\underline{v^0}$	$\mathsf{ret}\ [\![v^0]\!]^\rho$
$e_1^0 e_2^0$	$\mathsf{do}\ x_1 \leftarrow [\![e_1^0]\!]^\rho; x_2 \leftarrow [\![e_2^0]\!]^\rho; x_1 x_2$
$\langle e^1 \rangle$	$[\![e^1]\!]^\rho$
$\mathsf{run}\ e^0$	$\mathsf{do}\ x \leftarrow [\![e^0]\!]^\rho; x\langle ? \rangle$
v^0	$[\![v^0]\!]^\rho$
x^0	e where $e = \rho(x^0)$
$\lambda x^0.e^0$	$\lambda x.[\![e^0]\!]^{\rho, x^0 : x}$
$\langle v^1 \rangle$	$\mathsf{b}(r)[\![v^1]\!]_r^\rho$
e^1	$[\![e^1]\!]^\rho$
$\underline{v^1}$	$\mathsf{ret}\ (\mathsf{b}(r)[\![v^1]\!]_r^\rho)$
$e_1^1 e_2^1$	$\mathsf{do}\ x_1' \leftarrow [\![e_1^1]\!]^\rho; x_2' \leftarrow [\![e_2^1]\!]^\rho; \mathsf{ret}\ (\mathsf{b}(r)\mathsf{do}\ x_1 \leftarrow x_1'\langle r \rangle; x_2 \leftarrow x_2'\langle r \rangle; x_1 x_2)$
$\lambda x^1.e^1$	$\nu X.\mathsf{do}\ x' \leftarrow [\![e^1]\!]^{\rho, x^1 : \mathsf{b}(r) r.X}; \mathsf{ret}\ (\mathsf{b}(r)\mathsf{ret}\ \lambda x.x'\langle r\{X : x\}\rangle)$
$\tilde{}e^0$	$[\![e^0]\!]^\rho$
v^1	$[\![v^1]\!]_\theta^\rho$
x^1	$\mathsf{ret}\ e[r : \theta]$ where $\mathsf{b}(r)e = \rho(x^1)$
$\lambda x^1.v^1$	$\mathsf{ret}\ \lambda x.[\![v^1]\!]_\theta^{\rho, x^1 : \mathsf{b}(r) x}$
$v_1^1 v_2^1$	$\mathsf{do}\ x_1 \leftarrow [\![v_1^1]\!]_\theta^\rho; x_2 \leftarrow [\![v_2^1]\!]_\theta^\rho; x_1 x_2$

Lemma 2 (Preservation of Evaluation). *For any $\mathcal{X}$ and E*

- $e^0 \subset\xrightarrow{0} v^0$ *implies* $(\mathcal{X} | [\![e^0]\!]^\rho, E) \stackrel{*}{\Longrightarrow} (\mathcal{X}' | \mathsf{ret}\ [\![v^0]\!]^\rho, E)$ *for some* $\mathcal{X}'$
- $e^1 \subset\xrightarrow{1} v^1$ *implies* $(\mathcal{X} | [\![e^1]\!]^\rho, E) \stackrel{*}{\Longrightarrow} (\mathcal{X}' | \mathsf{ret}\ \mathsf{b}(r)[\![v^1]\!]_r^\rho, E)$ *for some* $\mathcal{X}'$

provided $x^1 \in \mathrm{FV}(e^n)$ *implies* $\rho(x^1) = \mathsf{b}(r)r.X$ *for some* $X \in \mathcal{X}$.

We conclude with three examples of MetaML_2- terms. Each term evaluates to the same MetaML_2-value $v^0 \stackrel{\Delta}{\equiv} \langle \lambda x^1.x^1 \rangle$, i.e. the code representing the object-level identity function. However, the MML_ν^N-translation of these terms reflect the different complexity of the evalution to v^0.

- The term $e_0^0 \stackrel{\Delta}{\equiv} \underline{v^0}$ evaluates immediately to v^0.
  The translation $[\![v^0]\!]^\emptyset$ is given by $e \stackrel{\Delta}{\equiv} \mathsf{b}(r)\mathsf{ret}\ \lambda x.\mathsf{ret}\ x$, where $\mathsf{ret}\ \lambda x.\mathsf{ret}\ x$ is the CBV monadic translation of $\lambda x.x$. The translation $[\![e_0^0]\!]^\emptyset$ is simply $\mathsf{ret}\ e$. Therefore, Theorem 6 holds trivially for $e_0^0 \subset\xrightarrow{0} v_0^0$.
- The term $e_1^0 \stackrel{\Delta}{\equiv} \langle \lambda x^1.\underline{x^1} \rangle$ evaluates to v^0, but the evaluation steps are more complex. This complexity is mirrored in the translation $[\![e_1^0]\!]^\emptyset$ given by

$e_1 \stackrel{\Delta}{\equiv} \nu X.\mathsf{do}\ x' \leftarrow \mathsf{ret}\ (\mathsf{b}(r)\mathsf{ret}\ r.X);$
$\mathsf{ret}\ (\mathsf{b}(r)\mathsf{ret}\ \lambda x.x'\langle r\{X\!:\!x\}\rangle)$

Theorem 6 for $e_1^0 \stackrel{0}{\hookrightarrow} v^0$ yields $(\emptyset|e_1,\Box) \stackrel{+}{\Longrightarrow} (X|\mathsf{ret}\ e,\Box)$.

– The term $e_2^0 \stackrel{\Delta}{\equiv} \langle\lambda x^1.\tilde{}((\lambda x^0.x^0)\underline{\langle x^1\rangle})\rangle$ evaluates to v^0, and requires evaluation within the body of the λx^1-binder. The translation $[\![e_2^0]\!]^\emptyset$ is given by

$e_2 \stackrel{\Delta}{\equiv} \nu X.\mathsf{do}\ x' \leftarrow \mathsf{do}\ x_1 \leftarrow \mathsf{ret}\ (\lambda x.\mathsf{ret}\ x);$
$x_2 \leftarrow \mathsf{ret}\ (\mathsf{b}(r)\mathsf{ret}\ r.X);$
$x_1 x_2;$
$\mathsf{ret}\ (\mathsf{b}(r)\mathsf{ret}\ \lambda x.x'\langle r\{X\!:\!x\}\rangle)$

Theorem 6 yields $(\emptyset|e_2,\Box) \stackrel{+}{\Longrightarrow} (X|\mathsf{ret}\ e,\Box)$, but the steps needed to reach the final configuration are strictly more than in the previous case.

7 Relating MML_ν^N to CMS

In this section we recall CMS [AZ02], a purely functional calculus of mixin modules, and introduce ML_Σ^N, a variant of MML_ν^N. Then we define a translation of CMS in ML_Σ^N preserving CMS typing and simplification up to Ariola's equational axioms [AB02] for recursion. We summarize the main differences between MML_ν^N and CMS (for those already familiar with CMS).

– CMS has a fixed infinite set of names (but a program uses only finitely many of them) and no fresh name generation facility.
– CMS is a pure calculus, thus we can restrict to the fragment of MML_ν^N without computational types, called ML^N.
– In CMS recursion is bundled in mixin, and removing it results in a very inexpressive calculus. On the contrary, ML^N is an interesting calculus (comparable to the λ-calculus) even without recursion, and one can add recursion following standard approaches.

7.1 CMS

We recall the calculus of mixin modules CMS, and refer to [AZ99, AZ02] for further details. The syntax of CMS is abstracted over symbolic names $X \in \mathsf{Name}$, and term variables $x \in \mathsf{X}$. For simplicity, we avoid to introduce core terms and types (in [AM04] the calculus is parametrized w.r.t. a core calculus).

– $\boxed{\tau \in \mathsf{CMST}::=[\Sigma_1;\Sigma_2]}$ types, where $\Sigma\!:\mathsf{Name} \stackrel{fin}{\to} \mathsf{CMST}$
– $\boxed{E \in \mathsf{CMSE}::=x \mid [\iota;o;\rho] \mid E_1+E_2 \mid E\setminus X \mid E!X \mid E.X}$ terms, where $\iota\!:\mathsf{X} \stackrel{fin}{\to} \mathsf{Name}$, $o\!:\mathsf{Name} \stackrel{fin}{\to} \mathsf{CMSE}$, $\rho\!:\mathsf{X} \stackrel{fin}{\to} \mathsf{CMSE}$ and we implicitly require that $\mathsf{dom}(\iota)\#\mathsf{dom}(\rho)$ for well-formed of $[\iota;o;\rho]$.

Free variables are defined as follows (omitting trivial cases): $\mathrm{FV}([\iota;o;\rho]) = (\mathrm{FV}(o)\cup\mathrm{FV}(\rho))\setminus(\mathsf{dom}(\iota)\cup\mathsf{dom}(\rho))$. Thus one can freely rename the bound

variables in $dom(\iota) \cup \mathsf{dom}(\rho)$, as done implicitly in the reduction rule (sum) below. We first give an informal overview of the calculus:

- The type $[\Sigma_1; \Sigma_2]$ specifies the names and types of the *deferred* (Σ_1) and *defined* (Σ_2) components of a mixin. The deferred components can be referred in the mixin, but are not defined, therefore they need to be resolved (see the freeze operation described below). The defined components corresponds to the exported definitions of the mixin.
- Term variables are used for local referencing of components, whereas names are needed for dealing with global access and linking of components. As in MML_ν^N, names are not terms.
- In a basic mixin $[\iota; o; \rho]$, ι specifies the deferred components. The mapping to names is needed for component resolution (see the freeze operation described below). The defined components (o) are associated with names, whereas local components (ρ) are introduced by variables and can be mutually recursive.
- The sum operation ($E_1 + E_2$) performs the union of the deferred components (in the sense that components with the same name are shared), and the disjoint union of the defined and local components of the two mixins.
- The *freeze* operation ($E!X$) binds the deferred component X to the expression of the defined component X in the same mixin; in this way a name can be resolved, and a deferred component becomes local. Cross-module recursion is obtained as a combination of the sum and the freeze operations.
- The *delete* operation ($E \setminus X$) is used for hiding defined components.
- Selection of a defined component ($E.X$) is only allowed for mixin with no deferred components.

Typing Rules. The typing judgment has form $\Gamma \vdash_{\mathsf{CMS}} E\colon\tau$, where $\Gamma\colon \mathsf{X} \xrightarrow{fin} \mathsf{CMST}$. The typing rules are given in Table 6, where two signatures Σ_1 and Σ_2 are *compatible* iff $\Sigma_1(X) = \Sigma_2(X)$ for all $X \in \mathsf{dom}(\Sigma_1) \cap \mathsf{dom}(\Sigma_2)$.

Simplification Rules. We define the relation $\xrightarrow[\mathsf{CMS}]{}$ as the compatible closure of the simplification rules defined in Table 7.

7.2 ML_Σ^N

The syntax of ML_Σ^N is defined in two steps. First, we remove from MML_ν^N computational types (and consequently monadic operations, like $\nu X.e$). In the resulting calculus, called ML^N, the computation relation disappears (CMS is a pure calculus), and $\mathcal{X}$ could be left implicit in the typing judgments $\mathcal{X}; \Pi; \Gamma \vdash e\colon\tau$, since the typing judgments of a derivation must use the same $\mathcal{X}$. Then we add records and mutual recursion:

- $\boxed{\tau \in \mathsf{T}_\mathcal{X} + = \Sigma}$ types, where $\Sigma \in \mathsf{\Sigma}_\mathcal{X} \triangleq \mathcal{X} \xrightarrow{fin} \mathsf{T}_\mathcal{X}$ is a $\mathcal{X}$-signature
- $\boxed{e \in \mathsf{E} + = o \mid e.X \mid e_1 + e_2 \mid e \setminus X \mid \mathsf{let}\ \rho\ \mathsf{in}\ e}$ terms, where

  $o\colon \mathsf{Name} \xrightarrow{fin} \mathsf{E}$ is a record $\{X_i\colon e_i | i \in m\}$ and

  $\rho\colon \mathsf{X} \xrightarrow{fin} \mathsf{E}$ is a (recursive) binding $\{x_i\colon e_i | i \in m\}$.

Table 6. Type System for CMS

$$\text{var}\ \frac{}{\Gamma \vdash_{\mathsf{CMS}} x\colon \tau}\ \Gamma(x) = \tau \qquad \text{delete}\ \frac{\Gamma \vdash_{\mathsf{CMS}} e\colon [\Sigma_1; \Sigma_2]}{\Gamma \vdash_{\mathsf{CMS}} e \setminus X\colon [\Sigma_1; \Sigma_2 \setminus X]}$$

$$\text{mixin}\ \frac{\{\Gamma, \Sigma_1 \circ \iota, \Gamma' \vdash_{\mathsf{CMS}} o(X)\colon \Sigma_2(X) \mid X \in \mathsf{dom}(o)\} \quad \{\Gamma, \Sigma_1 \circ \iota, \Gamma' \vdash_{\mathsf{CMS}} \rho(x)\colon \Gamma'(x) \mid x \in \mathsf{dom}(\rho)\}}{\Gamma \vdash_{\mathsf{CMS}} [\iota; o; \rho]\colon [\Sigma_1; \Sigma_2]}\ \begin{array}{l}\mathsf{dom}(\Gamma') = \mathsf{dom}(\rho)\\ \mathsf{dom}(\Sigma_1) = \mathsf{img}(\iota)\\ \mathsf{dom}(\Sigma_2) = \mathsf{dom}(o)\end{array}$$

$$\text{sum}\ \frac{\Gamma \vdash_{\mathsf{CMS}} e_1\colon [\Sigma_1^1; \Sigma_2^1] \quad \Gamma \vdash_{\mathsf{CMS}} e_2\colon [\Sigma_1^2; \Sigma_2^2]}{\Gamma \vdash_{\mathsf{CMS}} e_1 + e_2\colon [\Sigma_1^1, \Sigma_1^2; \Sigma_2^1, \Sigma_2^2]}\ \begin{array}{l}\Sigma_1^1 \text{ compatible with } \Sigma_1^2\\ \mathsf{dom}(\Sigma_2^1) \# \mathsf{dom}(\Sigma_2^2)\end{array}$$

$$\text{freeze}\ \frac{\Gamma \vdash_{\mathsf{CMS}} e\colon [\Sigma_1; \Sigma_2]}{\Gamma \vdash_{\mathsf{CMS}} e!X\colon [\Sigma_1 \setminus X; \Sigma_2]}\ \tau = \Sigma_1(X) = \Sigma_2(X)$$

$$\text{select}\ \frac{\Gamma \vdash_{\mathsf{CMS}} e\colon [\emptyset; \Sigma]}{\Gamma \vdash_{\mathsf{CMS}} e.X\colon \tau}\ \tau = \Sigma(X)$$

Table 7. Simplification rules for CMS

sum) $[\iota_1; o_1; \rho_1] + [\iota_2; o_2; \rho_2] \xrightarrow[\mathsf{CMS}]{} [\iota_1, \iota_2; o_1, o_2; \rho_1, \rho_2]$ if $\mathsf{dom}(o_1) \# \mathsf{dom}(o_2)$

delete) $[\iota; o; \rho] \setminus X \xrightarrow[\mathsf{CMS}]{} [\iota; o_{\setminus X}; \rho]$

freeze) $[\iota, \{x\colon X\}; o, \{X\colon E\}; \rho]!X \xrightarrow[\mathsf{CMS}]{} [\iota; o, \{X\colon E\}; \rho, \{x\colon E\}]$

select) $[; o, \{X\colon E\}; \rho].X \xrightarrow[\mathsf{CMS}]{} E[x\colon [; X\colon \rho(x); \rho].X \mid x \in \mathsf{dom}(\rho)]$

Free variables are defined as follows (omitting trivial cases): $\mathrm{FV}(\mathsf{let}\ \rho\ \mathsf{in}\ e) = \mathrm{FV}(e) \setminus \mathsf{dom}(\rho)$.

The type $\Sigma \equiv \{X_i\colon \tau_i | i \in m\}$ classifies records of the form $\{X_i\colon e_i | i \in m\}$, i.e. with a fixed set of components. Notice that records should not be confused with resolvers. In particular, a fragment of type $[\Sigma|\tau]$ can be linked with a resolver of any signature $\Sigma' \supseteq \Sigma$. The operations on records correspond to the CMS primitives for mixins: $e.X$ selects the component named X, $e_1 + e_2$ concatenates two records (provided their component names are disjoint), and $e \setminus X$ removes the component named X (if present). The let construct allows mutually recursive declarations, which are used to encode the local components of a CMS module. The order of record components and mutually recursive declarations are immaterial, therefore o and ρ are not sequences but functions (with finite domain).

Table 8 gives the typing rules for the new constructs. The properties of the type system in Section 4 extend in the obvious way to ML^{N}_{Σ}. We define

Table 8. Additional Typing Rules for ML^N_Σ

$$o\ \frac{\{\mathcal{X};\Pi;\Gamma \vdash e_i{:}\,\tau_i \mid i \in m\}}{\mathcal{X};\Pi;\Gamma \vdash \{X_i{:}\,e_i | i \in m\}{:}\,\{X_i{:}\,\tau_i | i \in m\}} \qquad \text{select}\ \frac{\Sigma(X) = \tau \quad \mathcal{X};\Pi;\Gamma \vdash e{:}\,\Sigma}{\mathcal{X};\Pi;\Gamma \vdash e.X{:}\,\tau}$$

$$\text{plus}\ \frac{\mathcal{X};\Pi;\Gamma \vdash e_1{:}\,\Sigma_1 \quad \mathcal{X};\Pi;\Gamma \vdash e_2{:}\,\Sigma_2}{\mathcal{X};\Pi;\Gamma \vdash e_1 + e_2{:}\,\Sigma_1,\Sigma_2}\ \mathsf{dom}(\Sigma_1)\#\mathsf{dom}(\Sigma_2)$$

$$\text{delete}\ \frac{\mathcal{X};\Pi;\Gamma \vdash e{:}\,\Sigma}{\mathcal{X};\Pi;\Gamma \vdash e \setminus X{:}\,\Sigma \setminus X}$$

$$\text{rec}\ \frac{\{\mathcal{X};\Pi;\Gamma,\Gamma' \vdash \rho(x){:}\,\Gamma'(x) \mid x \in \mathsf{dom}(\rho)\} \quad \mathcal{X};\Pi;\Gamma,\Gamma' \vdash e{:}\,\tau}{\mathcal{X};\Pi;\Gamma \vdash \mathsf{let}\ \rho\ \mathsf{in}\ e{:}\,\tau}\ \mathsf{dom}(\Gamma') = \mathsf{dom}(\rho)$$

simplification $\longrightarrow$ for ML^N_Σ as the compatible closure of the simplification rules for MML^N_ν (see Section 4.2) and the following simplification rules for record operations and mutually recursive declarations:

select) $o.X \longrightarrow e$ if $e \equiv o(X)$
plus) $o_1 + o_2 \longrightarrow o_1, o_2$ if $\mathsf{dom}(o_1)\#\mathsf{dom}(o_2)$
delete) $o \setminus X \longrightarrow o_{\setminus X}$
unfolding) $\mathsf{let}\ \rho\ \mathsf{in}\ e \longrightarrow e[x{:}\,\mathsf{let}\ \rho\ \mathsf{in}\ \rho(x) \mid x \in \mathsf{dom}(\rho)]$

Simplification for ML^N_Σ enjoys confluence (Theorem 1) and subject reduction (Theorem 2).

7.3 Translation of CMS into ML^N_Σ

The key idea of the translation consists in translating a mixin type $[\Sigma_1;\Sigma_2]$ in $[\Sigma'_1|\Sigma'_2]$, in this way we obtain a compositional translation of CMS terms. In contrast, a translation based on functional types, where $[\Sigma_1;\Sigma_2]$ is translated in $\Sigma'_1 \to \Sigma'_2$, is not compositional (the problem is in the translation of $e_1 + e_2$, which must be driven by the type of e_1 and e_2).

Table 9 gives the translation of CMS in ML^N_Σ. The translation can be easily extended to core terms (see [AM04]).

In the translation of a basic mixin $[\iota;o;\rho]$ the deferred variables x ($x \in \mathsf{dom}(\iota)$) are replaced with the resolution $r.X$ of the corresponding name $X = \iota(x)$, whereas the local variables x ($x \in \mathsf{dom}(\rho)$) are bound by the let construct for mutually recursive declarations. (A similar translation would not work in $\nu^\square$, because of the limitations in typing discussed in Section 8).

The translation of selection $E.X$ uses the empty resolver ?, since in CMS selection is allowed only for mixins without deferred components.

The freeze operator $E!X$ resolves a deferred component X with the corresponding output component. This resolution may introduce a recursive definition, since the output component X could be defined in terms of the corre-

Table 9. Translation of CMS in ML^N_Σ

CMS typing	ML^N_Σ typing
$\Gamma \vdash_{\mathsf{CMS}} E:\tau$	$\mathcal{X};\emptyset;\Gamma' \vdash E':\tau'$
CMS type	ML^N_Σ type
$[\Sigma_1;\Sigma_2]$	$[\Sigma'_1\vert\Sigma'_2]$
CMS term	ML^N_Σ term
x	x
$[\iota;o;\rho]$	$\mathsf{b}(r)(\mathsf{let}\ \rho'\ \mathsf{in}\ o')[x:r.X \mid \iota(x)=X]$
E_1+E_2	$\mathsf{b}(r)E'_1\langle r\rangle + E'_2\langle r\rangle$
$E\setminus X$	$\mathsf{b}(r)E'\langle r\rangle \setminus X$
$E.X$	$E'\langle ?\rangle.X$
$E!X$	$\mathsf{b}(r)\mathsf{let}\ \{x_1:x_2.X, x_2:E'\langle r\{X:x_1\}\rangle\}\ \mathsf{in}\ x_2$

the translations of Γ, Σ, o and ρ are defined pointwise.

sponding deferred component. Therefore, the translation defines the record x_2 by resolving the name X with the X component of the record x_2 itself.

The typing preservation property of the translation can be proved

Theorem 7 (Typing preservation). *If $\Gamma \vdash_{\mathsf{CMS}} E:\tau$, then $\mathcal{X};\emptyset;\Gamma' \vdash E':\tau'$, where $\mathcal{X}$ includes all names occurring in the derivation of $\Gamma \vdash_{\mathsf{CMS}} E:\tau$.*

The translation preserves also the semantics of CMS, but this can be proved only up to some equational axioms for mutually recursive declarations

$$\mathsf{C}[\mathsf{let}\ \rho\ \mathsf{in}\ e] = \mathsf{let}\ \rho\ \mathsf{in}\ \mathsf{C}[e] \qquad \text{(lift)}$$
$$\mathsf{let}\ \rho_1\ \mathsf{in}\ \mathsf{let}\ \rho_2\ \mathsf{in}\ e = \mathsf{let}\ \rho_1,\rho_2\ \mathsf{in}\ e \qquad \text{(ext-merge)}$$
$$\mathsf{let}\ \rho_1, x:(\mathsf{let}\ \rho_2\ \mathsf{in}\ e_2)\ \mathsf{in}\ e_1 = \mathsf{let}\ \rho_1,\rho_2, x:e_2\ \mathsf{in}\ e_1 \qquad \text{(int-merge)}$$
$$\mathsf{let}\ \rho, x:e_1\ \mathsf{in}\ e_2 = \mathsf{let}\ \rho[x:e_1]\ \mathsf{in}\ e_2[x:e_1]\ \text{if}\ x \notin \mathrm{FV}(e_1) \qquad \text{(sub)}$$

The (lift) axiom corresponds to Ariola's lift axioms, in principle it can be instantiated with any ML^N_Σ context $\mathsf{C}[\,]$, but for proving Theorem 8 it suffices to consider the following contexts: $\boxed{\mathsf{C}[\,]::=\Box+e \mid e+\Box \mid \Box\setminus X \mid \Box.X}$.

The (ext-merge) and (int-merge) axioms are Ariola's merge axioms, whereas (sub) is derivable from Ariola's axioms.

Let R denotes the set of the three axioms above, and S denotes the set of equational axioms corresponding to the simplification rules for ML^N_Σ; then the translation is proved to preserve the CMS simplification up to $=_{S\cup R}$ (i.e. the congruence induced by the axioms in $S\cup R$).

Theorem 8 (Semantics preservation). *If $E_1 \xrightarrow[\mathsf{CMS}]{} E_2$, then $E'_1 =_{S\cup R} E'_2$.*

The translation of the non-recursive subset of CMS (i.e. no local declarations ρ and no freeze $E!X$) is a lot simpler, moreover its simplifications are mapped to plain ML^N_Σ simplifications.

We conclude this section with some examples of CMS reductions and their translations in ML^N_Σ.

Example 4. Consider the **plus** reduction $E_1 \xrightarrow[\mathsf{CMS}]{} E_2$, where
$E_1 \equiv [\{x\colon A\}; \{R\colon x\}; \emptyset] + [\emptyset; \{A\colon y\}; \emptyset]$,
$E_2 \equiv [\{x\colon A\}; \{A\colon y, R\colon x\}; \emptyset]$.
The translations of E_1 and E_2 are given by
$E_1' \equiv \mathsf{b}(r)(\mathsf{b}(r_1)(\mathsf{let}\ \emptyset\ \mathsf{in}\ \{R\colon r_1.A\}))\langle r\rangle + (\mathsf{b}(r_2)\mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y\})\langle r\rangle$
$E_2' \equiv \mathsf{b}(r)\mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon r.A\}$.
By repeatedly applying simplifications and equational axioms in R we get
$E_1' \xrightarrow{*}$ (by **link**)
$\mathsf{b}(r)(\mathsf{let}\ \emptyset\ \mathsf{in}\ \{R\colon r.A\}) + \mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y\} =_R$ (by **lift**)
$\mathsf{b}(r)\mathsf{let}\ \emptyset\ \mathsf{in}\ \mathsf{let}\ \emptyset\ \mathsf{in}\ \{R\colon r.A\} + \{A\colon y\} =_R$ (by **ext-merge**)
$\mathsf{b}(r)\mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon r.A\} \equiv E_2'$.

Example 5. Consider the **freeze** reduction $E_3 \xrightarrow[\mathsf{CMS}]{} E_4$, where
$E_3 \equiv E_2!A$ with E_2 defined as in the previous example,
$E_4 \equiv [\emptyset; \{A\colon y, R\colon x\}; \{x\colon y\}]$.
The translations of E_3 and E_4 are given by
$E_3' \equiv \mathsf{b}(r)\mathsf{let}\ \{x_1\colon x_2.A, x_2\colon E_2'\langle r\{A\colon x_1\}\rangle\}\ \mathsf{in}\ x_2$
$E_4' \equiv \mathsf{b}(r)\mathsf{let}\ \{x\colon y\}\ \mathsf{in}\ \{A\colon y, R\colon x\}$.
By repeatedly applying simplifications and equational axioms in R we get
$E_3' \longrightarrow$ (by **link**)
$\mathsf{b}(r)\mathsf{let}\ \{x_1\colon x_2.A, x_2\colon \mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon (r\{A\colon x_1\}).A\}\}\ \mathsf{in}\ x_2 \longrightarrow$ (by **resolve**)
$\mathsf{b}(r)\mathsf{let}\ \{x_1\colon x_2.A, x_2\colon \mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon x_1\}\}\ \mathsf{in}\ x_2 =_R$ (by **sub**)
$\mathsf{b}(r)\mathsf{let}\ \{x_1\colon (\mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon x_1\}).A\}\ \mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon x_1\} =_R$ (by **ext-merge**)
$\mathsf{b}(r)\mathsf{let}\ \{x_1\colon (\mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon x_1\}).A\}\ \mathsf{in}\ \{A\colon y, R\colon x_1\} =_R$ (by **lift**)
$\mathsf{b}(r)\mathsf{let}\ \{x_1\colon \mathsf{let}\ \emptyset\ \mathsf{in}\ \{A\colon y, R\colon x_1\}.A\}\ \mathsf{in}\ \{A\colon y, R\colon x_1\} =_R$ (by **int-merge**)
$\mathsf{b}(r)\mathsf{let}\ \{x_1\colon \{A\colon y, R\colon x_1\}.A\}\ \mathsf{in}\ \{A\colon y, R\colon x_1\} \longrightarrow$ (by **resolve**)
$\mathsf{b}(r)\mathsf{let}\ \{x_1\colon y\}\ \mathsf{in}\ \{A\colon y, R\colon x_1\}$ which is α-equivalent to E_4'.

Example 6. Consider the **select** reduction $E_5 \xrightarrow[\mathsf{CMS}]{} E_6$, where
$E_5 \equiv E_4.R$ with E_4 defined as in the previous example,
$E_6 \equiv [\emptyset; \{X\colon y\}; \{x\colon y\}].X$.
The translations of E_5 and E_6 are given by
$E_5' \equiv (\mathsf{b}(r)\mathsf{let}\ \{x\colon y\}\ \mathsf{in}\ \{A\colon y, R\colon x\})\langle ?\rangle.R$,
$E_6' \equiv (\mathsf{b}(r)\mathsf{let}\ \{x\colon y\}\ \mathsf{in}\ \{X\colon y\})\langle ?\rangle.X$.
By repeatedly applying simplifications we get
$E_5' \longrightarrow$ (by **link**)
$(\mathsf{let}\ \{x\colon y\}\ \mathsf{in}\ \{A\colon y, R\colon x\}).R \longrightarrow$ (by **unfolding**)
$\{A\colon y, R\colon \mathsf{let}\ \{x\colon y\}\ \mathsf{in}\ y\}.R \longrightarrow$ (by **resolve**)
$\mathsf{let}\ \{x\colon y\}\ \mathsf{in}\ y \longrightarrow$ (by **unfolding**)
y.

Analogously, $E_6' \xrightarrow{*} y$, therefore $E_5' =_S E_6'$. Unlike the other examples, there is no way to get from E_5' to E_6' by applying simplifications and equations axioms in R. This explains the use of $=_{S\cup R}$ in stating Theorem 8.

8 Conclusions and Related Work

This section compares MML_ν^N with the CBV calculi FreshML and $\nu^\Box$. First, we recall briefly the main features of these two calculi, then we make a critical assessment based on a comparison with MML_ν^N.

- FreshML of [SPG03][2] is an extension of ML, based on a solid mathematical theory [GP99], that provides a convenient support for meta-programming. Namely, in FreshML abstract representations (i.e. modulo α-conversion) of object-level syntax co-exist with pattern matching facilities (similar to those usable on concrete parse trees) to analyse these representations.
- $\nu^\Box$ of [Nan02, NPar] is a refinement of $\lambda^\Box$ [DP01], which provides better support for symbolic manipulation by exploiting some features of FreshML (the calculus presented in [NPar] does not support program analysis). The stated aim is to combine safely the best features of $\lambda^\Box$ (the ability to execute closed code) and $\lambda^\bigcirc$ [Dav96] (the ability to manipulate open code). MetaML has similar aims, but adopts the opposite strategy, i.e. it starts from $\lambda^\bigcirc$, nor does it build on top of FreshML (names are not part of the MetaML syntax).

If we ignore the different styles for describing the operational semantics of FreshML (a CBV evaluation relation), $\nu^\Box$ (a CBV small-step reduction relation) and MML_ν^N (a simplification and a computation relation), the key differences are:

- FreshML supports program transformation, in particular the analysis of object-level programs represented as values of an inductive datatype involving *abstraction* types $\langle \mathsf{name} \rangle \tau$.
- $\nu^\Box$ of [NPar] supports only program generation, (object-level) programs e of type τ with unresolved names in $\mathcal{X}$ are represented by values of type $\Box_{\mathcal{X}}\tau$. The typing rules for $\Box_{\mathcal{X}}\tau$ are fairly restrictive, because $\mathcal{X}$ has to include **all** unresolved names in e (and e should not contain free variables x).
- MML_ν^N supports only program generation, (object-level) programs e of type τ abstracted w.r.t. a name resolver r of signature Σ are represented by terms of type $[\Sigma|\tau]$. The typing rules for $[\Sigma|\tau]$ are similar to those for a functional type $\Sigma \to \tau$, in particular e may use other name resolvers besides r.

MML_ν^N *versus FreshML.* Typing judgments of FreshML have the standard format $\Gamma \vdash e{:}\,\tau$, because names do not occur in types (and in particular there are no resolvers and no signatures for typing resolvers).

In FreshML names are terms (and there is a type name of names), so generation of a fresh name is denoted by $\nu x.e$, where x is a term variable which gets bound to the fresh name, and e is the term where the fresh name can be used. In MML_ν^N names occur both in types and in terms, and using x in place of a name X would entail a type system with dependent types (which would be problematic), thus we must use a different binder $\nu X.e$ for names. FreshML, unlike MML_ν^N, supports the manipulation of object-level syntax modulo α-conversion. This is possible because FreshML has:

[2] There is another version of FreshML [PG00] with a more elaborate type system, which is able to mask the computational effects due to generation of fresh names.

- an equality type name of names
- abstraction types $\langle \mathsf{name} \rangle \tau$ classifying equivalence classes of pairs (X, e) modulo renaming of X with names fresh for e (of type τ), terms $\langle e_1 \rangle e_2$ to form name abstractions, and patterns $\langle x \rangle p$ to deconstructed them.
- a name swapping operation, which is crucial (in combination with name generation) to define the operational semantics of name abstraction matching.

It should be possible to extend MML_ν^N with these features of FreshML. However, one should keep a clear distinction between the types $\langle \mathsf{name} \rangle \tau$ and $[\Sigma|\tau]$. The first type classifies representations of object-level syntax modulo α-conversion, while the second classifies fragments modulo simplification, thus it cannot support program analysis.

MML_ν^N *versus* $\nu^\Box$. Typing judgments of $\nu^\Box$ take the form $\Sigma; \Delta; \Gamma \vdash e : \tau[\mathcal{X}]$, where $\mathcal{X} \subseteq \mathsf{dom}(\Sigma)$ includes the names occurring *free* in e, and Δ has declarations of the form $u_i : \tau_i[\mathcal{X}_i]$ with $\mathcal{X}_i \subseteq \mathsf{dom}(\Sigma)$.

In $\nu^\Box$ the type of a name X is fixed at name generation time. This is a bad name space management policy, which goes against common practice in programming language design (e.g. of modules systems). MML_ν^N follows the approach of mainstream module languages, where different modules can assign to the same name different types (and values). Therefore, programming in $\nu^\Box$ forces an overuse of name generation, because the language restricts name reuse.

In $\nu^\Box$ terms includes names, so our $\theta.X$ is replaced by X, in other words there is a *default resolver* which is left implicit. Linking $u\langle\Theta\rangle$ uses a function $\Theta \equiv \langle X_i \to e_i | i \in m \rangle$ to modify the default resolver. The typing judgments for explicit substitutions Θ take the form $\Sigma; \Delta; \Gamma \vdash \Theta : \mathcal{X}[\mathcal{X}']$, where $\mathcal{X}'$ includes the names *used* by the modified resolver to resolve the names in $\mathcal{X}$, e.g. $\mathcal{X} \subseteq \mathcal{X}'$ when Θ is empty. The following explicit substitution principle is admissible

$$\frac{\Sigma; \Delta; \Gamma \vdash \Theta : \mathcal{X}[\mathcal{X}'] \quad \Sigma; \Delta; \Gamma \vdash e : \tau[\mathcal{X}]}{\Sigma; \Delta; \Gamma \vdash e[\Theta] : \tau[\mathcal{X}']}$$

Our type $[\Sigma|\tau]$ corresponds to $\Box_{\mathcal{X}}\tau$ with $\mathcal{X} = \mathsf{dom}(\Sigma)$. Typing rules for $\Box_{\mathcal{X}}\tau$ are related to those for necessity of S4 modal logic, e.g. $\Box_{\mathcal{X}}\tau$ introduction is

$$\frac{\Sigma; \Delta; \emptyset \vdash e : \tau[\mathcal{X}]}{\Sigma; \Delta; \Gamma \vdash \mathsf{box}\, e : \Box_{\mathcal{X}}\tau[\mathcal{X}']}$$

This rule is very restrictive: it forbids having free term variables x in e, and acts like an *implicit binder* for the free names X of e (i.e. it binds the default resolver for e). Without these restrictions substitution would be unsound in the type system of $\nu^\Box$. Such restrictions have no reason to exist in MML_ν^N, because we allow multiple name resolvers, and fragments $\mathsf{b}(r)e$ are formed by abstracting over one name resolver. Furthermore, making name resolvers explicit, avoid the need to introduce *non-standard* forms of substitution.

The observations above are formalized by a CBV translation $_'$ of $\nu^{\Box}$-terms[3] into MML_{ν}^{N}, where the resolver variable r corresponds to the default resolver, which is implicit in $\nu^{\Box}$.

$e \in \nu^{\Box}$	$e' \in \mathsf{MML}_{\nu}^{N}$	$e \in \nu^{\Box}$	$e' \in \mathsf{MML}_{\nu}^{N}$
x	$\mathsf{ret}\ x$	X	$r.X$
$\lambda x{:}\tau.e$	$\mathsf{ret}\ (\lambda x.e')$	$u\langle X_i \to e_i\rangle$	$u\langle r\{X_i{:}\, e'_i\}\rangle$
$e_1\ e_2$	$\mathsf{do}\ x_1 \leftarrow e'_1; x_2 \leftarrow e'_2; x_1 x_2$	$\mathsf{box}\ e$	$\mathsf{ret}\ (\mathsf{b}(r)e')$
$\nu X{:}\tau.e$	$\nu X.e'$	$\mathsf{letbox}\ u = e_1\ \mathsf{in}\ e_2$	$\mathsf{do}\ u \leftarrow e'_1; e'_2$

We do not define the translation on types and assignments, since in $\nu^{\Box}$ the definition of well-formed signatures $\Sigma \vdash$ and types $\Sigma \vdash \tau$ is quite complex.

In conclusion, the key novelty of MML_{ν}^{N} is to make name resolvers explicit and to allow a multiplicity of them, as a consequence we gain in simplicity and expressivity. Moreover, by building on top of a fairly simple form of extensible records, we are better placed to exploit existing programming language implementations (like O'Caml).

References

[AB02] Z. M. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Annals of pure and applied logic*, 117(1-3):95–178, 2002.

[AM04] D. Ancona and E. Moggi. A fresh calculus for names management. In Karsai and Visser [KV04].

[AZ99] Davide Ancona and Elena Zucca. A primitive calculus for module systems. In *Proc. Int'l Conf. Principles & Practice Declarative Programming*, volume 1702 of *Lecture Notes in Computer Science*, pages 62–79. Springer, 1999.

[AZ02] D. Ancona and E. Zucca. A calculus of module systems. *J. Funct. Programming*, 12(2):91–132, March 2002. Extended version of [AZ99].

[BCT02] D. Batory, C. Consel, and W. Taha, editors. *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*. Springer, October 2002.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *Conf. Rec. POPL '97: 24th ACM Symp. Princ. of Prog. Langs.*, pages 266–277, 1997.

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CM94] L. Cardelli and J. C. Mitchell. Operations on records. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 295–350. The MIT Press, Cambridge, MA, 1994.

[CMS03] C. Calcagno, E. Moggi, and T. Sheard. Closed types for a safe imperative MetaML. *J. Funct. Programming*, 13(3):545–571, 2003.

[3] In [NPar] the operational semantics (and the typing) of $\nu X.e$ differs from that adopted by (us and) FreshML. To avoid unnecessary complications, we work as if $\nu^{\Box}$ is FreshML compliant.

[CMT04] C. Calcagno, E. Moggi, and W. Taha. ML-like inference for classifiers. In *Programming Languages & Systems, 13th European Symp. Programming*, volume 2986 of *Lecture Notes in Computer Science*. Springer, 2004.

[CTHL03] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In Krzysztof Czarnecki, Frank Pfenning, and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering (GPCE)*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[Dav96] R. Davies. A temporal-logic approach to binding-time analysis. In *the Symposium on Logic in Computer Science (LICS '96)*, pages 184–195, New Brunswick, 1996. IEEE Computer Society Press.

[DP01] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, 2001.

[ESOP00] *Programming Languages & Systems, 9th European Symp. Programming*, volume 1782 of *Lecture Notes in Computer Science*. Springer, 2000.

[GP99] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax involving binders. In *Proc. 14th Ann. IEEE Symp. Logic in Comput. Sci.*, pages 214–224, July 1999.

[GS04] J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*. Wiley Publishing Inc, 2004.

[HW03] Christian Haack and J. B. Wells. Type error slicing in implicitly typed, higher-order languages. In *Programming Languages & Systems, 12th European Symp. Programming*, volume 2618 of *Lecture Notes in Computer Science*, pages 284–301. Springer, 2003. Superseded by [HW04].

[HW04] Christian Haack and J. B. Wells. Type error slicing in implicitly typed, higher-order languages. *Sci. Comput. Programming*, 50:189–224, 2004. Supersedes [HW03].

[KV04] G. Karsai and E. Visser, editors. *Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*. Springer, October 2004.

[LBCO04] Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors. *Domain-Specific Program Generation*. Number 3016 in Lecture Notes in Computer Science. Springer-Verlag, 2004.

[Met01] MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from `http://www.cs.rice.edu/~{}taha/MetaOCaml/`, 2001.

[MF03] E. Moggi and S. Fagorzi. A monadic multi-stage metalanguage. In *Proc. FoSSaCS '03*, volume 2620 of *Lecture Notes in Computer Science*. Springer, 2003.

[MT00] Elena Machkasova and Franklyn A. Turbak. A calculus for link-time compilation. In ESOP '00 [ESOP00], pages 260–274.

[Nan02] Aleksandar Nanevski. Meta-programming with names and necessity. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP-02)*, ACM SIGPLAN notices, New York, October 2002. ACM Press.

[NPar] A. Nanevski and F. Pfenning. Staged computations with names and necessity. *J. Funct. Programming*, to appear.

[PG00] Andrew M. Pitts and Murdoch J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Proc. Mathematics of Program Construction, 5th Int'l Conf. (MPC 2000)*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255, Ponte de Lima, Portugal, July 2000. Springer.

[PS03] F. Pfenning and Y. Smaragdakis, editors. *Generative Programming and Component Engineering*, volume 2830 of *Lecture Notes in Computer Science*. Springer, September 2003.

[She01] T. Sheard. Accomplishments and research challenges in metaprogramming. In W. Taha, editor, *Proc. of the Int. Work. on Semantics, Applications, and Implementations of Program Generation (SAIG)*, volume 2196 of *LNCS*, pages 2–46. Springer-Verlag, 2001.

[SPG03] Mark R. Shinwell, Andrew M. Pitts, and Murdoch J. Gabbay. FreshML: Programming with binders made simple. In *Proc. 8th Int'l Conf. Functional Programming*. ACM Press, 2003.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming, 2nd Edition*. Addison Wesley, 2002.

[Tah99] W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Inst. of Science and Technology, 1999. Available from `ftp://cse.ogi.edu/pub/tech-reports/README.html`.

[TN03] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, 2003.

[TS97] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM)*, pages 203–217, Amsterdam, 1997. ACM Press.

[WV99] J. B. Wells and René Vestergaard. Confluent equational reasoning for linking with first-class primitive modules (long version). A short version is [WV00]. Full paper, 3 appendices of proofs, August 1999.

[WV00] J. B. Wells and René Vestergaard. Equational reasoning for linking with first-class primitive modules. In ESOP '00 [ESOP00], pages 412–428. A long version is [WV99].

Assertion-Based Encapsulation, Object Invariants and Simulations

David A. Naumann*

Department of Computer Science,
Stevens Institute of Technology, Hoboken, NJ 07030, USA

Abstract. In object-oriented programming, reentrant method invocations and shared references make it difficult to achieve adequate encapsulation for sound modular reasoning. This tutorial paper surveys recent progress using auxiliary state (ghost fields) to describe and achieve encapsulation. Encapsulation is assessed in terms of modular reasoning about invariants and simulations.

1 Introduction

This paper addresses two problems in reasoning about sequential object-oriented programs in languages like Java: reentrant callbacks and sharing of mutable objects. We present an approach to modular reasoning based on the addition of ghost ("fictitious", auxiliary) fields with which intended structural relationships can be expressed —in particular, dependency relationships. The difficulties have various manifestations in informal practice but can be understood most clearly in terms of formal reasoning, in particular reasoning about object invariants and simulation relations. Object invariants are essential for modular proof of correctness and simulations are essential for modular proof of equivalence or refinement of class implementations. The approach offers interdependent solutions to the two problems. It was developed initially by Barnett et al. [5] and has been extended by Leino, Müller, and others [27, 7, 40, 43].

Besides giving a tutorial introduction to the approach, we compare it with other approaches and suggest possible extensions and opportunities for future work. Müller et al. [36] and Jacobs et al. [24] give good introductions to the problems and other solution approaches. The book by Szyperski et al. [52] illustrates the problems using more realistic examples than can fit in a research paper. We assume the reader has minimal familiarity with Java-like languages; some familiarity with design patterns [22] may be helpful.

Outline. Section 2 sketches the problems. Section 3 addresses invariants and reentrant callbacks in detail and how they are handled in the ghost variable approach. Section 4 addresses invariants and object sharing using a notion of ownership. Section 5 discusses additional issues concerning ownership-based invariants and Section 6 shows how the approach can be used with simulations. Section 7 considers extension of the approach to invariants that depend on non-owned objects —a discipline for friendly cooperation. Section 8 discusses prospects and challenges for further extensions.

* Supported in part by the US National Science Foundation (CCR-0208984 and CCF-0429894) and by Microsoft.

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 251–273, 2005.
© Springer-Verlag Berlin Heidelberg 2005

2 How Shared Objects and Reentrant Callbacks Violate Encapsulation

Several constructs in Java and similar programming languages are intended to provide encapsulation. A *package* collects interrelated classes and serves as a unit of scope. Each instance of a *class* provides some abstraction, as simple as a complex number or as complex as a database server. A *method specification* describes an operation in terms of the abstraction. A method *implementation* uses other abstractions and is verified, for the sake of modularity, with respect to their specifications.

Less frequently, a class itself provides some abstraction, represented using static fields.[1] More frequently, instances of multiple classes collectively provide an abstraction of interest, e.g., a collection and its iterators. In this paper we focus on the abstraction provided by a single instance or small group of instances.

To show that a method implementation satisfies its specification it is often essential to reason in terms of an *object invariant*[2] for the target or receiver object $\mathbf{self}$. An object's invariant involves consistency conditions on internal data structures —its representation, made up of so-called *rep objects*— and the connection with the abstraction they represent. To a first approximation, an object's invariant is an implicit precondition and postcondition for every method [23]. More precisely, it is not suitable to be visible to clients and is maintained solely by the methods of the object's class since, owing to encapsulation, it is not susceptible to interference by client code.

These notions are clear and effective in situations where abstractions are composed by hierarchical layering. As we shall explain, however, both reentrant callbacks and object sharing can violate simple hierarchical structure.

Reentrant Callbacks. Consider some kind of sensor playing the role of Subject in the Observer pattern [22]. The sensor maintains a set of registered Views: when the sensor value reaches the threshhold, $v.thresh$, of a given view v, the sensor invokes method $v.notify()$ and removes v from the set. This description is in terms of a set, part of the abstraction offered by the Subject; the implementation might store views in an array ordered by $thresh$ values. The pattern cannot be seen simply as a client layered upon an abstraction, because $notify$ is an *upcall* to the client. The difficulty is that $v.notify$ may make a *reentrant callback* to the sensor. Consider the following sequence of invocations, where s is a sensor: A client or asynchronous event invokes $s.update()$ which changes the value of the sensor. Before returning, $update$ invokes $v.notify()$ where v is a view registered with s. Now v maintains a reference, $v.sensor$, to s and in order for $notify$ to do its job it invokes $v.sensor.getval()$ to determine the current sensor value. Because $v.sensor = s$, this invocation of $getval$ is known as a *reentrant callback* as control returns to s while another method invocation (here $update$) is in progress.

It is common that a reentrant callback is intended. In the example, $getval$ might simply read a field and cause no problem. However, trouble is likely if v invokes on s a

[1] Associated with a class rather than with each instance.

[2] In a class-based language it is natural to include in a class the declaration of an invariant, with the interpretation that each instance satisfies the invariant. To emphasize the instance-oriented nature of such invariants, we use the term *object invariant* although some authors prefer "class invariant".

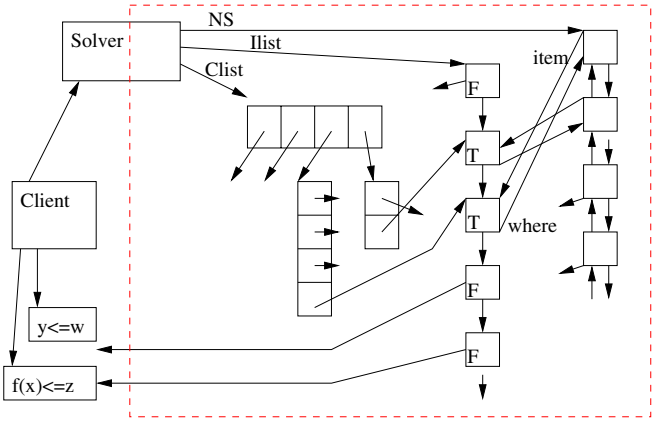


Fig. 1. An example data structure

method $enum$ that enumerates the current set of views of s, since $enum$ likely depends on the invariant that the array of views is in a consistent state. In terms of the abstract interface, is v still in the set of registered views? In terms of the array, is v in fact in the array?

This example involves cyclic linking $s \rightarrow v \rightarrow s$ of heap objects. We consider next a different problem due to shared references.

Shared Mutable Objects. An illustration of the challenging invariants found in object-oriented programs is the structure of objects and references in Fig. 1. This depicts the data structures used in a constraint solving algorithm [49]. Several structural invariants must be maintained by $Solver$ for correctness and efficiency of the algorithm. For example, the objects in the vertical column on the far right form a doubly linked list rooted at NS and their $item$ fields point to elements of the list rooted at $Ilist$. Moreover, each of those items is in the range of the array of arrays $Clist$. And there is cross-linking between $Ilist$ and NS. These examples can be written as follows:[3]

$$
\begin{array}{l}
(NS = \mathbf{null} \vee NS.prev = \mathbf{null}) \\
\wedge\, (\,\forall\, p \in NS.next^* \mid (p.next = \mathbf{null} \vee p.next.prev = p) \\
\qquad\qquad\qquad \wedge\, p.item \in Ilist.next^* \wedge (\,\exists\, x, j \mid Clist[x][j] = p.item\,) \\
\qquad\qquad\qquad \wedge\, p.item.where = p\,)
\end{array}
$$

The client program is intended to have a reference to the $Solver$ object, and the data structure includes pointers to client objects that represent constraints (e.g., "$y \leq w$"). The latter pointers go against a strict hierarchical layering of abstractions (clients using solvers), but this is not necessarily a problem. But there is no reason for clients to have references to the objects within the dashed boundary; these are intended to comprise the encapsulated representation of the solver. A reference to one of these rep objects would

[3] Here $*$ denotes reflexive transitive closure of field dereferences. We use a Java-like notation with implicit dereferencing: $p.next$ is the value of field $next$ in the object pointed to by p.

```
class Subject{
  private x, y : int := 0, 1;   view : View := ...;
  invariant I^Subject   where I^Subject =df (self.x < self.y)
  method m() { x := x + 1; view.notify(); y := y + 1; }
  method f() : float { return 1/(y − x); }
}
class View {
  z : Subject := ...;
  method notify() { ... z.f(); }
}
```

Fig. 2. Simple example of reentrant callback. (Occurrence of a field name like x without qualifier abbreviates $\mathbf{self}.x$.)

be problematic because the client could update the object and falsify the invariant of $Solver$.

Suppose for simplicity that class $Solver$ has no proper subclasses or superclasses, other than $Object$ (which is reasonable in this example since the class basically provides a single algorithm). Fields NS, $Ilist$, and $Clist$ can be given private scope so no code of other classes can update them. We can reason in a modular way about invariants that depend only on these fields, e.g., $Clist \neq \mathbf{null}$. If such an invariant is established by the constructor then —absent reentrant callbacks— we can assume it as a precondition of every method of $Solver$ so long as it is established as a postcondition of every method of $Solver$.

The formula displayed above, however, depends on other objects; scope-based encapsulation does not protect them from interference by client code. For example, if the client held a reference o to the first node in NS, i.e., $o = NS$ with $NS \neq \mathbf{null}$, then it could set $o.prev := o$, violating the first line of the invariant which enforces acyclicity. If a method of $Solver$ is then invoked, it is not sound to assume the invariant as a precondition.

A notion of heap encapsulation that fits this situation is *ownership*. The idea has three parts. First, the objects comprising the representation of an instance of $Solver$ are considered to be owned by it —the encapsulation boundary encloses exactly the owned objects. Second, the invariant is only allowed to depend on owned objects. Third, invariant-falsifying updates are prevented by some means. The most common means is to disallow references to owned objects from outsiders. This is the *dominator property* [16]: Every path to a rep of $Solver$ s from an object that is not a rep of s must go through s. Ownership is the topic of Section 4.

3 Reentrance and Object Invariants

In this section we set aside ownership and present a discipline for invariants in the presence of reentrant callbacks.

For clarity we use the contrived example in Fig. 2. In class $Subject$, the object invariant $x < y$ is established by initialization $x, y := 0, 1$. Method f relies on the invariant (to avoid division by 0); it maintains the invariant because it does no updates.

At first glance, the invariant is also maintained by m since it increments x and y by the same amount. Because x and y are local, they are not susceptible to update in code outside of class $Subject$ —and this is what we need for modular reasoning about $Subject$. But there is the possibility of a reentrant callback. For object s of type $Subject$, an invocation of $s.m$ results in the invocation $s.view.notify$ in a state where the declared invariant does not hold for s. Now $s.view.notify$ in turn invokes $z.f$, so if $s.view.z = s$ then an error occurs. If instead $notify$ invoked $z.m$ then the program would diverge due to nonterminating recursion.

Possible Solutions. One reaction to the example is to disallow any reentrant callback. This could be done using static analysis for control flow, taking into account aliasing in order not to disallow too many programs. Such analyses are usually not modular, however. A specification of allowed calling patterns might also be required, since for example if f simply returned x then the callback $m \rightarrow notify \rightarrow f$ is harmless and possibly desirable.

The problem is similar to interference found in concurrent programs and one might try to solve it using locks. But here we are concerned with a single thread of control; if a lock was taken by the initial call to m and that lock prevented a reentrant call then deadlock would result. (In Java, a lock held by a given thread does *not* prevent that thread from reentering the object, precisely to avoid deadlock.) A related solution is to introduce a boolean field inm to represent that a call of m is in progress and to use $\neg inm$ as precondition of m and f. This has similarities to the approach advocated later.

Another approach to the problem is to require the invariant to hold prior to any method call, lest that call lead to a reentrant callback. This has been advocated in the literature [29, 32] and is sometimes called *visible state semantics* [36]. Our example can be revised to fit this discipline, by changing the body of m to this:

$$x := x + 1;\ y := y + 1;\ view.notify(); \tag{1}$$

Note that $\mathcal{I}^{Subject}$ holds after the second assignment so it is sound for $view.notify$ to rely on it, e.g., by making reentrant calls. But this approach does not scale to more realistic programs, where the invariant may involve several data structures, update of which is done by method calls. Most method calls do not lead to reentrant callbacks and we already noted that some reentrant callbacks are harmless, even desirable.

Another alternative would be to state the invariant as an explicit precondition for those methods that depend on it. Then $notify$ in the example would be rejected because it could not establish the precondition for $z.f()$. This alternative must be rejected on grounds of information hiding, the predicate $\mathcal{I}^{Subject}$, like most invariants, depends on internals that should be encapsulated within the class.

Various techniques have been proposed to hide information about an invariant while expressing that it is in force. One alternative is to introduce a typestate [20] to stand for "the invariant is in force". Another approach is to treat its name as opaque with respect to its definition [8], as may be done in higher order logic using existential quantification [9]. Another way to treat the invariant as an opaque predicate, which to the author's knowledge has not been explored, is to use a pure method [26] to represent the invariant; this could be of practical use in runtime verification and hiding of internals could be achieved using visibility rules of the programming language.

The Boogie Approach. The best features of the preceding alternatives are combined in the so-called Boogie approach of Barnett et al. [5]. The idea is to make explicit in preconditions not the invariant predicate, e.g., $\mathcal{I}^{Subject}$, but rather a boolean abstraction of it (similar to the typestate approach). For reasons that will become clear, we use the term *inv/own discipline* for the Boogie approach.

To a first approximation, the discipline uses a *ghost* (auxiliary) field inv of type boolean so that $o.inv$ represents the condition that "the invariant $\mathcal{I}[o/\mathbf{self}]$ is in force".[4] The idea is that the implication $o.inv \Rightarrow \mathcal{I}[o/\mathbf{self}]$ can be made to hold in every state, while $\mathcal{I}[o/\mathbf{self}]$ itself is violated from time to time for field updates. The idea can be used with an ordinary field, but here we use a ghost field that has no runtime significance but rather is used only for reasoning. Field inv is considered to be public and declared in the root class $Object$, so $\mathbf{self}.inv$ can appear as a precondition of any method that depends on the invariant. For our running example, both methods f and m would have precondition $\mathbf{self}.inv$.

```
class Subject {
  private x, y : int := 0, 1;
  private view : View := ...;
  invariant I^Subject
    where I^Subject =_df self.x < self.y
  method m()
    requires self.inv
    ensures self.inv
  { unpack self;
    assert ¬self.inv
    x := x + 1;
    view.notify(self);
    y := y + 1; }
}
class View {
  method notify(Subject z) {z.m();}
}
```

Fig. 3. Variation on Fig. 2 (incomplete)

The discipline imposes several proof obligations in order to ensure that the following is a *program invariant*, i.e., it holds in all reachable states:

$$(\,\forall\, o \mid\; o.inv \;\Rightarrow\; \mathcal{I}[o/\mathbf{self}]\,) \tag{2}$$

Consider a method m with (at least) precondition $\mathbf{self}.inv$. To reason about correctness of an implementation of m, within the scope where $\mathcal{I}$ is visible, the conjunction of (2) and $\mathbf{self}.inv$ yield $\mathcal{I}$. On the other hand, outside the scope a reasoner sees only the precondition $\mathbf{self}.inv$.

To emphasize that inv is a ghost variable used only for reasoning, the discipline uses special commands **pack** and **unpack** to set inv true and false, respectively. Key proof obligations are imposed on these. The obligations are most easily understood in terms of allowed proof outlines. In particular, certain preconditions are stipulated for the special commands and for field updates. The two most important obligations are:

- The precondition for **pack** E is $\neg E.inv \wedge \mathcal{I}[E/\mathbf{self}]$. Clearly $\mathcal{I}[E/\mathbf{self}]$ is necessary to maintain (2) upon truthification of $E.inv$. The first conjunct prevents reentrance to this region of code since **pack** sets $E.inv$ true.
- The precondition for a field update $E.f := E'$ is $\neg E.inv$. This ensures that the update does not falsify (2) for the object E.

[4] Here $[o/\mathbf{self}]$ denotes substitution of o for $\mathbf{self}$, taking aliasing into account.

Consider the code in Fig. 3, a variation on Fig. 2. Here the $Subject$ passes **self** as an argument to $notify$ and an incomplete annotation is sketched. The update $x := x + 1$, which abbreviates $\mathbf{self}.x := \mathbf{self}.x + 1$, is subject to precondition $\neg\mathbf{self}.inv$. As the precondition of m is $\mathbf{self}.inv$, the special command **unpack self** is needed to set inv false. Now we consider some options in reasoning about $notify$.

One possibility is for $z.inv$ to be a precondition for $notify$. Then the implementation of $notify$ is correct: according to the specification of m, the call $z.m()$ has precondition $z.inv$. The implementation of $Subject.m$ is thus forced to establish $\mathbf{self}.inv$ preceding the call to $notify$.

```
unpack self;
assert ¬self.inv
x := x + 1;
y := y + 1;
assert I^Subject
pack self;
view.notify(self);
```

Setting $\mathbf{self}.inv$ true is the effect of the special command **pack self**, but the stipulated precondition for **pack self** is $\mathcal{I}$ and this assertion would not hold immediately following the update $x := x + 1$. The situation can be repaired as in the code at right, where, as in (1), the assignment $y := y + 1$ precedes invocation of $notify$ so that the implementation of m can be verified. This seems satisfactory for the example but in general it is impractical to impose the visible state semantics for invariants. The example does show that the discipline can handle this pattern of reasoning.

Another possibility is to retain the implementation of m, so that inv is only restored at the end, as in the code on the right. For the call to $notify$ to be correct, $notify$ cannot have precondition $z.inv$. But then the implementation of $notify$ is not correct, because it has no way to establish $z.inv$ which is the precondition for $z.m$. On the other hand, $notify$ is free to invoke on z any method that does not require $z.inv$.

```
unpack self;
assert ¬self.inv
x := x + 1;
view.notify(self);
y := y + 1;
assert I^Subject
pack self;
```

Summary. Through use of ghost field inv following the rules of the discipline, a harmful reentrant callback can be prevented while allowing some callbacks. There is a clear intuition, that $z.inv$ stands for "z is in a consistent state" (it is *packed*, for short). Yet the internal representation of $Subject$ is not exposed to $View$; there is no need for predicate $\mathcal{I}^{Subject}$ to be visible outside $Subject$.

4 Sharing and Object Invariants

Let us set aside the issue of reentrance and consider another toy example, now involving shared references (see Fig. 4). The initialization of $Subject2$ establishes $\mathcal{I}^{Subject2}$. The annotation of m is correct: The first assertion follows from precondition $\mathbf{self}.inv$ and program invariant (2). The second assertion follows from the first by straightforward reasoning about $incr$. Method $leak$ does no updates and thus maintains $\mathcal{I}^{Subject2}$.

Unfortunately, $main$ uses $leak$ to falsify (2). In a state where $s.inv$ is true, and thus $\mathcal{I}^{Subject2}[s/\mathbf{self}]$ by (2), $main$ uses $s.leak()$ to obtain a (shared) reference i to $s.x$. The invocation $i.incr()$ then updates the val field, falsifying $s.x.val < s.y.val$ and thus falsifying $s.inv \Rightarrow \mathcal{I}^{Subject2}[s/\mathbf{self}]$.

```
class Integer { public val : int;
  method incr() { val := val + 1; }
}
class Subject2 {
  private x : Integer := new Integer(0);
  private y : Integer := new Integer(1);
  invariant I^Subject2   where I^Subject2 =df (x ≠ null ≠ y ∧ x.val < y.val)
  method m()
    requires self.inv
    ensures self.inv
  { unpack self;
    assert I^Subject2; x.incr(); y.incr(); assert I^Subject2;
    pack self; }
  method leak() : Integer { return x; }
}
class Main{   s : Subject2 := new Subject2; ...
  method main() { i : Integer := s.leak(); i.incr(); s.m(); }
}
```

Fig. 4. Invariant dependent on rep objects

One diagnosis is that the invariant of $Subject2$ should not be allowed to depend on fields of objects other that **self**. Indeed, some proposals in the literature on invariants for object-oriented programs are only sound under this restriction [30]. But for many classes this is highly impractical. For an example, consider updates to the structure of the solver in Section 2, assuming list operations are coded in object-oriented style, e.g., using methods of the node classes for setting fields and for recursive list operations.

The name "*leak*" indicates our diagnosis: just as field x is private, so too the object referenced by x belongs within class $Subject2$. More precisely, it is a *rep* object — part of the representation of an abstraction provided by an instance of $Subject2$. A rep belongs to its owner and this licenses its owner's invariant to depend on it. Thus the programming discipline must prevent updates of reps by code outside $Subject2$.

Ownership. As mentioned in Section 2, some ownership systems prevent harmful updates by preventing the existence of references from client to rep (the dominator property that all paths to a rep go through its owner). It is easy to violate the dominator property: a method could return a rep pointer, or pass one as an argument to a client method.

The dominator property can be enforced using a type system such as the Universe system [35] and variations on Ownership Types [17, 13, 12, 1]. These systems do not directly enforce the dominator property, which is expressed in terms of paths. Rather, they constrain references, disallowing any object outside an ownership domain from having a pointer to inside the domain. This means that from the point of view of a particular object s, the heap can be partitioned into three blocks:

- the singleton containing just s
- the objects owned by s (which, together with s, are called an *island*)
- all other objects

In these terms, the invariant for s is only allowed to depend on fields of objects in the island of s.

The name "*leak*" suggests that what has gone wrong in the example is the very existence of a shared reference. Ownership type systems prevent harmful updates by alias control: static rules would designate that x is owned and would reject method $leak$. This approach has attractive features but it has proved difficult to find an ownership type system that admits common design patterns and also enforces sufficiently strong encapsulation for modular reasoning about object invariants. In particular, many examples call for the transfer of ownership (see Section 5) which does not sit well with type-based systems. Moreover ownership typing involves rather special program annotations (decorating declarations with ownership information).

The alternative presented below controls *uses* of references and represents ownership restrictions with assertions.[5]

Before turning to that topic we note that Separation Logic [50] provides a way to express that a predicate depends on only some objects in the heap (and correctness assertions that express on what part of the heap the correctness of a command depends). The logic has been used for encapsulating dependence of invariants in simple imperative programs [41] but in its current form the logic depends on a concrete view of heap cells in which all fields are explicit. This is at odds with subtyping and inheritance which affords modular reasoning about extensible classes. This is an exciting line of research, but adoption of a nonstandard logic for specification and verification has significant cost.

Ownership Using Ghost Fields. The first step is quite direct. Each object has ghost field own to point to its owner. If an object o currently has no owner (as is the case when initially constructed), $o.own = \mathbf{null}$. An object encapsulates the objects it transitively owns. We define transitive ownership as a relation on references as follows: $o \succ p$ iff either $o = p.own$ or $o \succ p.own$. Note that $\succ$ is state-dependent. The invariant, $\mathcal{I}^C$, for a class C is considered *admissible* just if whenever $\mathcal{I}_C$ depends on $p.f$ for some object p then either $\mathbf{self} = p$ or $\mathbf{self} \succ p$.

In virtue of representing the ownership relation by a ghost pointer to the owner, we have imposed the invariant that an object has at most one owner. Transitive ownership thus imposes a hierarchical structure on the heap —though one that is mutable.[6]

Rather than preventing aliases to encapsulated reps from clients, the inv/own discipline prevents updates that falsify the invariant. For invariants that depend only on fields of $\mathbf{self}$, this was achieved by imposing on every update $E.f := E'$ the precondition $\neg E.inv$. It would be sound, but hardly practical, to impose now the precondition

$$\neg E.inv \wedge (\forall\, o \mid o \succ E \;\Rightarrow\; \neg o.inv\ )$$

so that no object with an invariant dependent on $E.f$ is packed. One reason this precondition is impractical is that the code performing the update of E would have to have ensured that many objects are unpacked, which hardly seems modular. Another reason

[5] Skalka and Smith [51] also study use-based object confinement, for different purposes.

[6] Because field own is mutable, it is possible to create a cycle of owners. But owing to the stipulated preconditions of the discipline, objects in a cycle cannot be packed.

is that if o owns p it makes no sense to unpack p unless o is already unpacked, since when it is packed o's invariant depends on p.

The idea with precondition $\neg E.inv$ for an update $E.f := E'$ is that E should get unpacked before updates are performed on it. This means in a sense that control crosses the encapsulation boundary for E. The discipline uses one more ghost field, $com : \mathbf{bool}$, in order to impose a discipline whereby the flow of control across encapsulation boundaries respects the current ownership hierarchy. The name stands for "committed": $o.com$ implies $o.inv$ but says in addition that o is committed to its owner and can only be unpacked after its owner gets unpacked. This idea is embodied in two additional program invariants:

$$(\forall\, o, p \mid o.inv \wedge p.own = o \;\Rightarrow\; p.com\ ) \tag{3}$$

$$(\forall\, o \mid o.com \;\Rightarrow\; o.inv\ ) \tag{4}$$

The key consequence of these invariants is the *transitive ownership lemma*: If $o \succ p$ and $\neg p.inv$ then $\neg o.inv$. It is now possible to maintain program invariant (2) simply by stipulating for every field update $E.f := E'$ the precondition $\neg E.inv$. If the update is made in a state where for some object o we have that $\mathcal{I}[o/\mathbf{self}]$ depends on $E.f$ then $o \succ E$ by admissibility of $\mathcal{I}$. And by the transitive ownership lemma, $\neg E.inv$ implies $\neg o.inv$.

We have prevented interference not by alias control nor by syntactic conditions but rather by a precondition, expressed in terms of auxiliary state that encodes dependency and hierarchy.

Typically, the precondition of a method that performs updates is $\mathbf{self}.inv \wedge \neg\mathbf{self}.com$. If it performs updates on a parameter x, an additional precondition will be $x.inv \wedge \neg x.com$. Manipulation of the com field is part of what it means to pack and unpack an object. For $\mathbf{unpack}\ E$, the stipulated precondition is now $E.inv \wedge \neg E.com$ and the effect[7] is

$$E.inv := \mathbf{false};\ \mathbf{foreach}\ o\ \mathbf{such\ that}\ o.own = E\ \mathbf{do}\ o.com := \mathbf{false};$$

For $\mathbf{pack}\ E$, the stipulated precondition is $\neg E.inv \wedge \mathcal{I}^C[E/\mathbf{self}] \wedge (\forall\, o \mid o.own = E \;\Rightarrow\; o.inv \wedge \neg o.com\ )$ where C is the type of E. The effect is

$$E.inv := \mathbf{true};\ \mathbf{foreach}\ o\ \mathbf{such\ that}\ o.own = E\ \mathbf{do}\ o.com := \mathbf{true};$$

5 Additional Aspects of the inv/own Discipline

The ingredients of the discipline are

- Assertions.
- Ghost fields.[8]

[7] The "**foreach**" part of the effect can be expressed using a specification statement: modifies com, ensures $(\forall\, o \mid (o.own = E \wedge \neg o.com) \vee (o.own \neq E \wedge o.own = \mathbf{old}(o.own))\ )$.

[8] With inv, own ranging over values that include class names, i.e., slightly beyond ordinary program data types. Similar use of class names is available in the JML specification via the **type** operator [26].

– Updates to ghost fields, including update of an unbounded number of objects (in **pack** E, for example, the com field of every object owned by E is updated).

This is quite limited machinery and thus the discipline is suitable for use in a variety of settings. It could be formalized within an ordinary program logic, most attractively a proof outline logic [45]. It is being explored in the context of Spec#, a tool based directly on a system of verification conditions, and in a tool developed by de Boer and Pierik [18]. In both cases the assertion language is (roughly) first order plus reachability but that is not essential.

Rather than relying entirely on annotations, practical use of the discipline can be streamlined through some simple abbreviations [5, 27]. A marked field declaration **rep** $f : T$ is syntactic sugar for the invariant $\mathbf{self}.f.own = \mathbf{self}$ and **peer** $f : T$ is syntactic sugar for $\mathbf{self}.f.own = \mathbf{self}.own$. It is also possible to infer, absent other annotation, that the implementation of a method with precondition $\mathbf{self}.inv$ should be annotated with **unpack self**; ...; **pack self** so this need not be written explicitly.

The discipline supports an attractive extension to frame conditions: without mention in a modifies clause, a method can update committed objects. For details see [5].

Quantification. We have formalized the program invariants (2–4) using quantifications that range over all allocated objects. Quantification is problematic. If quantification ranges over currently allocated objects then a quantified formula can be falsified by garbage collection, e.g., $(\exists\, o \mid P(o))$ is falsified if the only object with property P gets collected. This cannot happen if P connects o with other objects via ordinary fields, e.g., $P(o) = (o.f = C.x)$ with x a static field of class C, as then o is not garbage. But the problem occurs with as simple a property as $y.f = 1$ if y is a ghost field. Garbage sensitivity has been studied in depth by Calcagno et al. [14]. A workable solution is to ignore garbage collection in program logic, so quantifications range over all objects that have been allocated.

This still leaves the possibility of falsification by allocating a new object. Pierik, de Boer, and Clarke [43] have explored, for example, the Singleton pattern [22] where one might want the invariant of $Singleton$ to be

$$(\forall\, p \mid \mathbf{type}(p) \leq Singleton \;\Rightarrow\; p = Singleton.it) \tag{5}$$

where it is a static field of class $Singleton$ and $\leq$ denotes subtype. (Recall that our quantifications range over non-null object references; we write $\mathbf{type}(o)$ for the dynamic type of an object.)

This problem can be taken into account by including in the definition of admissibility that an invariant cannot be falsifiable by construction of new objects. The authors of the Boogie papers [5, 27] intend that invariants use quantification only over owned objects, which achieves this effect. Barnett and Naumann [40] impose it explicitly in their definition of admissibility.

An alternative is for predicates like (5) to be considered admissible and to stipulate a suitable precondition for object construction (**new**). This alternative has been worked out by Pierik et al. [43] based on a notion of update guard that we discuss in Section 7.

Ownership Transfer. A useful feature of the inv/own discipline is that, while it imposes hierarchical structure on the heap, that structure is mutable. Field own is initially **null**; a fresh object has no owner. The field is updated by special command

set-owner E **to** E', the effect of which is simply $E.own := E'$. As with ordinary field update, it is subject to precondition $\neg E.inv$. Moreover, in the case that $E' \neq \mathbf{null}$ the command adds to the objects owned by E' —and it adds to those transitively owned by the transitive owners of E'. Their invariants depend on their owned objects so we require them to be unpacked. The stipulated precondition for **set-owner** E **to** E' is $\neg E.inv \wedge (E' = \mathbf{null} \vee \neg E'.inv)$. Thus the ownership structure can change dynamically when the relevant invariants are not in force.

Change in ownership structure is difficult or impossible with ownership type systems, in part because the type system imposes the ownership conditions as a program invariant, i.e., true in every state. Transfer has been found to be useful in a number of situations. The most common seems to be initialization by the client of an abstraction that then becomes owned by another; this was pointed out by Leino and Nelson [21] with the example of a lexer that owns an input stream but that stream is constructed by the client. Transfer between peer owners is appropriate, for example, with several queues of tasks that are moved between queues for load balancing. The trickiest form of transfer is when an encapsulated rep is released to clients; this form has been highlighted by O'Hearn in the example of a memory allocator, considering that the allocator owns elements of the free list [41]. Other examples can be found in [27, 4].

Taking Subclasses into Account. If C is a subclass of D then an instance of C has fields of D and of C. Moreover, it should maintain the invariant, $\mathcal{I}^D$, of D but C may impose an additional invariant $\mathcal{I}^C$. Instead of using a boolean to track whether "the" invariant is in effect, the general form of the inv/own discipline lets inv range over classnames, with the interpretation that $o.inv \leq C$ means that o is packed with respect to the invariant of C and of any superclasses of C. This works smoothly if we assume $\mathcal{I}^{Object}$ is $\mathbf{true}$.

Owned objects are now owned at a particular class, i.e., field own ranges over $\mathbf{null}$ and pairs (C, o) with $\mathbf{type}(o) \leq C$ indicating that the object is owned by o at class C and is part of the representation on which $\mathcal{I}^C$ depends.

The **pack** and **unpack** commands are revised to mention the class involved. For **unpack** E **from** C, the stipulated precondition is now $E.inv = C \wedge \neg E.com$ and the effect is

$$E.inv := super(C);\ \mathbf{foreach}\ o\ \mathbf{such\ that}\ o.own = (E, C)\ \mathbf{do}\ o.com := \mathbf{false};$$

For **pack** E **to** C, the stipulated precondition is $E.inv = super(C) \wedge \mathcal{I}^C[E/\mathbf{self}] \wedge (\forall\, o \mid o.own = E \Rightarrow o.inv \wedge \neg o.com\ )$ and the effect is

$$E.inv := C;\ \mathbf{foreach}\ o\ \mathbf{such\ that}\ o.own = (E, C)\ \mathbf{do}\ o.com := \mathbf{true};$$

The program invariants are also adapted slightly.

$$(\forall\, o, C \mid o.inv \leq C \Rightarrow \mathcal{I}^C[o/\mathbf{self}]\ )$$
$$(\forall\, o, p, C \mid o.inv \leq C \wedge p.own = (o, C) \Rightarrow p.com\ )$$
$$(\forall\, o \mid o.com \Rightarrow o.inv \leq \mathbf{type}(o)\ )$$

Methods are dynamically dispatched, which raises the question how to express the precondition that before was just $inv = \mathbf{true}$. The Boogie paper [5] introduces notation

```
class Subject2 { //Alternate version
  private rep x : Integer := new Integer(0);
  private rep z : int := 0;
  invariant I^Subject2'   where I^Subject2' =df 0 ≤ z
  method m() { x.incr(); }   }
```

Fig. 5. Revised $Subject2$

which at a method call site means $E.inv = \mathbf{type}(E)$ but in the method implementation means that $\mathbf{self}.inv$ equals the static type. This is worked out by treating method inheritance as an abbreviation for a stub method with appropriate **unpack** and **pack**; this generates a proof obligation on an inherited method.

Soundness and Completeness. Soundness of the discipline is taken to mean that the three displayed conditions hold in every reachable state of a *properly annotated program*, i.e., one in which every field update and every instance of a special command **pack**, **unpack**, or **set-owner** is preceded by an assertion that implies the stipulated precondition.

For sequential programs in a Java-like language, soundness is sketched in the original Boogie paper [5] and more rigorously in [40]; see also [27]. Extension of the discipline to concurrent programs has also been investigated [25].

Completeness is another matter. It is not at all clear to this author how to formulate an interesting notion of completeness. Clearly it should be relative to completeness of an underlying proof system. The discipline hinges on having every object invariant expressed in the form $inv \Rightarrow \mathcal{I}$ with $\mathcal{I}$ admissible. Does completeness say that every predicate of this form that is in fact invariant can be shown so in a proof outline following the discipline? Related questions are which admissible predicates are expressible as formulas and which formulas denote admissible predicates. A convincing notion of completeness would be especially useful if it could be adapted to other disciplines like the one discussed in Section 7.

In what sense are invariants necessary at all? One could perhaps simply conjoin (2), (3), and (4) to preconditions and postconditions throughout the program. But this raises another expressiveness question. And for modularity it might require abstraction from internals, e.g., using model fields. Notions of completeness that take modularity into account have recently been studied by Pierik and de Boer [44].

Static Invariants. We have focused on object invariants that depend on instance fields. It is also sensible for an object invariant to depend on static fields, e.g., the Singleton invariant (5). There is also the possibility of a static invariant for a class. Examples are given by Leino and Müller [28] and by Pierik et al. [43]. The basic idea is to use a static field in the same way as inv, to represent whether the invariant of a class is in force. There are intricacies due to the way in which classes are initialized in Java.

6 Pointer Confinement and Simulation

Fig. 5 shows an alternate implementation of class $Subject2$ from Fig. 4. The behavior of the two versions is the same (at the level of abstraction of the programming language,

e.g., ignoring speed and size of object code). The standard way to prove behavioral equivalence of two modular units such as classes is by means of a coupling relation that has the simulation property. A *coupling* relates states for one implementation with states for the other. For an instance s of $Subject2$ in the first version (Fig. 4) and s' for the second version (Fig. 5), a suitable coupling is

$$s.x.val = s'.x.val \wedge s.y.val - s.x.val = s'.z$$

Such a relation is a *simulation* provided that it is preserved by corresponding method implementations —as it is by the two versions of m in the example. (The same technique is also used to prove refinement: in case one implementation diverges less often or is less nondeterministic, the notion of preservation is adapted slightly.) The technique is practical because the simulation property only needs to be proved for the reimplemented methods: For arbitrary program contexts, simulation should follow from simulation for the revised class, by a general *representation independence* property of the language.

The technique was articulated by Hoare [23] drawing on work of Milner [33] and has seen much development for use with purely functional programs [48, 34, 47] as well as first order imperative and concurrent programs [31]. For first order imperative programs the topic is thoroughly surveyed in the textbook by de Roever et al. [19]. Object oriented programs have features in common with higher order imperative programs, for which representation independence is nontrivial owing to semantic difficulties [42, 46, 39]. Two sources of complication in object oriented programs are inheritance and the ubiquitous use of recursive classes; these were addressed by Cavalcanti and the author [15] —under the drastic simplification that copying is used instead of sharing. Their results have been used to validate laws of program refactoring [11, 10].

The representation independence property, i.e., the possibility of reasoning in a modular way using simulations, is a measure of the encapsulation facilities of a language. We have seen how reentrant callbacks and heap sharing pose a challenge for encapsulation in object oriented programs. Using a static analysis for alias control in order to impose an ownership structure just for the class under revision, Banerjee and Naumann [2] are able to show representation independence for a rich imperative fragment of Java with class-based visibility, inheritance and dynamic binding, type casts and tests, recursive types, etc. A key feature of this work is the notion of *local coupling* which is a binary relation not on complete program states but just on a fragment of the heap consisting of a single instance of the class under revision together with its reps. That is, a local coupling relates pairs of islands. This induces a coupling relation for the entire program state.

There are two main shortcomings to the work of Banerjee and Naumann [2]. First, ownership transfer is disallowed by their confinement rules. Second, the result is inadequate for programs with callbacks because it is in terms of the standard notion of simulation: for method m to preserve the coupling means that if two states are initially coupled, then running the two versions of the implementation of m leads to coupled states. Recall that representation independence says, with A the class for which two versions are considered, that if all methods m of class A have the simulation property then the relation is preserved when those methods are used in arbitrary program contexts. In fact the proof obligation is not simply that m preserves the coupling, but rather

that it preserves the coupling *under the hypothesis that any method m invokes preserves the coupling*.[9] This assumption can be useful in establishing the simulation property for m, but only if the two implementations make the same method call and from a state where the coupling holds. But at intermediate steps in paired invocations of (the two versions of) m, the coupling relation need not hold —essentially for the same reason as invariants need not hold during updates of local state. The hypothesis is of no help if a client method is invoked at an intermediate step where the coupling does not hold.

It turns out that the inv/own discipline, which is concerned with preservation of invariants, can be adapted to simulations, i.e., preservation of coupling relations; see [4]. The intuition is that a coupling is just an invariant over two copies of program state. Moreover, field inv is observable (by specifications), so both versions of a method m of A have the same $\mathbf{unpack/pack}$ structure, so the coupling can take the form of an implication with antecedent inv. This form of coupling can then hold at intermediate points in m, in particular at method calls —so the hypothesis is now of use.

The adaptation is not trivial because the inv/own discipline only controls updates. Recall the example *leak* in Section 4. If we revise it as follows, so that the leaked reference is only read, then the program is compatible with the inv/own discipline.

$$\begin{array}{l}\mathbf{class}\ \mathit{Main}\ \{\quad s : \mathit{Subject}2 := \mathbf{new}\ \mathit{Subject}2;\\ \quad \mathbf{method}\ \mathit{main}()\ \{\ i : \mathit{Integer} := s.\mathit{leak}();\ \mathit{Print}(i.\mathit{val});\ \}\quad \}\end{array}$$

For invariants, it is only a problem if i is updated. But for simulations, we need independence from reps —not even dependence by reading— as otherwise a client's behavior can be affected and the representation is not fully encapsulated. This can be achieved by stipulating additional preconditions for field access [4] (which in practice can usually be discharged trivially in virtue of standard visibility rules).

Informal considerations of information hiding suggests that clients should not read fields of reps, and this is confirmed by the analysis of representation independence. In this regard it seems that the main advantage of the inv/own discipline over ownership types is the ability to temporarily violate the ownership property in order to transfer objects between owners. The Spec# project [6] is exploring inference to determine where the **set-owner**, **pack** , and **unpack** commands are needed. Integration with ownership types merits investigation.

7 Beyond Single-Object Invariants

At the beginning of Section 2 we focused attention on situations where each instance of a class is intended to provide some cohesive abstraction such as a collection. Such examples are ubiquitous, but so too are situations where several objects cooperate to provide some abstraction.

One example is iterators. To equip a Collection with the possibility of enumerating its elements, a separate object is instantiated for each enumeration. These Iterator

[9] The reason this is sound is similar to the justification for proof rules for recursive procedures: it is essentially the induction step for a proof by induction on the maximum depth of the method call stack.

objects need access to the internal data structure of the Collection, to get elements of the Collection and to track whether the Collection has changed in a way that makes the Iterator inconsistent and unusable.

One can imagine formulating a single invariant that pertains to the collective state of a Collection and its Iterators, but it is not clear with what program structure this invariant would be associated. Perhaps the iterator and Collection classes could be put in a single module, but associating the invariant with the module does not reflect that the natural unit is a single Collection instance together with its iterators.

An alternative using more familiar notions is to express the conditions in the object invariant for an Iterator. But it is not feasible for an Iterator to own the Collection on which it depends, since Iterators serve as part of the interface to clients. Aldrich and Chambers [1] explore a flexible notion of ownership type where the dominator property is not necessarily imposed, but absent this property it is not clear what modular reasoning is supported.

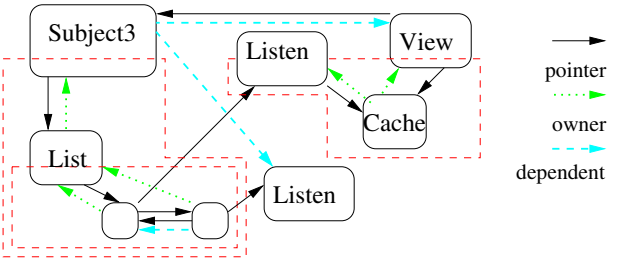
The need for object invariants to depend on non-owned objects arises in quite simple situations. In Section 2 we considered the $Solver$ invariant that involves doubly-linked list conditions $p.next = \mathbf{null} \vee p.next.prev = p$ for all p in $NS.next^*$. It is possible to associate the entire invariant with class $Solver$, but at the cost of a quantifier and reasoning about reachability. A less centralized formulation would push some of the conditions into object invariants for the rep objects, e.g., each node could maintain the invariant $next = \mathbf{null} \vee next.prev = \mathbf{self}$. But for this to be admissible, a node would need to own its successor. Such an ownership structure is workable for acyclic doubly-linked lists but not for cyclic ones (and awkward if iteration is used instead of recursion).

As a more elaborate example, consider the variation on the Observer pattern depicted in Fig. 6, where a separate Listener object is the target of the $notify$ callback. The dashed and dotted arrows are explained in due course. Dashed rectangles are used as before to indicate ownership encapsulation. In this arrangement it would seem that both the Listener and the View need to read and update their shared Cache object. The situation is similar to that for Collections/Iterators. We return to this point later. The next point to consider is that we aim to specify that notifications are required: the Subject has a version number that is incremented each time it is updated, and $notify$ brings the View back in sync. For simplicity we treat the state of the Subject as an integer —see the code in Fig. 6. The View maintains a copy of the state of the sensor, with its version number, in its Cache object. View also maintains the invariant that this version is not more than one step behind. We assume it is untenable for the View to own its Subject sbj. So this invariant is inadmissible according to the previous definition, because it depends on fields val and vsn of sbj.

A prerequisite for this dependence is of course that those fields are visible in $View$. Rather than giving them public visibility, let us suppose that $Subject3$ includes an explicit declaration

$$\mathbf{friend}\ View\ \mathbf{reads}\ vsn, val;$$

to extend the scope of visibility. The intention is not only to broaden the scope (as little as possible) but also to license dependence of $\mathcal{I}^{View}$ on these fields. It is thereby also signalled to $Subject3$ that it has a proof obligation: if s has type $Subject3$ then updates



class *Subject3* { val : **int**; vsn : **int**; $listeners$: $List\langle Listener\rangle$;
...}
class *Cache* { vsn : **int**; val : **int**; }
class *View* { sbj : $Subject$; **rep** st : $Cache$;
invariant $\mathcal{I}^{View}$ where $\mathcal{I}^{View} =_{df}\ sbj.vsn - 1 \leq st.vsn \leq sbj.vsn$
$\wedge(st.vsn = sbj.vsn \Rightarrow st.val = sbj.val)$;
...}
class *Listener* { st : $Cache$;
method $notify()$ { ... } }

Fig. 6. Observer pattern using separate listeners

to $s.val$ and $s.vsn$ must not falsify the invariant of any object v of type $View$ that is dependent on s.

Visibility Based Invariants. Müller and others [35, 27, 36] have worked out sound rules for reasoning about invariants in this sort of *peer* relationship. They use the term *visibility based invariant*, in contrast to ownership based invariants. In our example, the idea would be that $Subject3$ is responsible to maintain any invariants visible to it. In some examples this is quite manageable, but in general there is a problem. The visibility based approach works at the level of classes. In reasoning about an update of a given instance s of $Subject$, one must consider the invariant of any $v : View$ since the invariant of $View$ depends on fields of $Subject$. The question is how the reasoner gets a handle on those objects, given that there can be many instances of $View$, dependent on many different instances of $Subject3$.

In the example at hand, $s.listeners$ is intended to hold references to all listeners for views dependent on s, so that they can be notified of updates. Suppose that listeners have a field $myview$ so that the views dependent on s are those in $s.listeners.next^*.myview$. Then it suffices to prove that updating $s.val$ or $s.vsn$ does not falsify the invariant of those views. Thus the precondition for $s.val := \ldots$ would say that the dependent views are unpacked:

$$\mathbf{self}.inv > Subject \wedge (\forall\, v \mid v \ \mathbf{in}\ s.listeners.next^*.myview \Rightarrow v.inv > View\ ) \quad (6)$$

It is certainly possible to establish this precondition. In order to update fields declared in $Subject$, s must be unpacked from $Subject$, so s is not committed. If the views have the same owner, they also are not committed and thus they can be unpacked if they are not already. But must they have the same owner?

Anyway, packing and unpacking is designed to embody hierarchical encapsulation. Here we are considering peers that are in some sense within the same encapsulation boundary. Moreover, to repack a view v, the $Subject$ would need to justify that $\mathcal{I}^{View}[v/\mathbf{self}]$ but why should $\mathcal{I}^{View}[v/\mathbf{self}]$ be visible in $Subject$?

The preceding challenges are not insurmountable using just standard proof rules and the visibility assumptions. But by using a ghost field to track the relevant dependencies, more localized reasoning can be achieved.

The Friendship Discipline. We posited temporarily that an instance s of $Subject3$ has access to its dependent views via $listener$ and $myview$, but in fact $Listener$ has no such field so Subject3 only has references to the $Listener$s on which it is supposed to invoke $notify$. For another example of such a situation, a long-lived Collection might have many associated Iterators. The Iterators could depend on a timestamp field in the collection, so an Iterator can be considered invalid if the collection gets updated. But there may be no reason for the Collection to maintain a list of its iterators. Instead of incurring a performance cost to maintain the list merely for the sake of reasoning, it can be stored in a ghost field.

The Friendship discipline [7] extends the inv/own discipline by adding a ghost field $deps$ to hold references to dependents (the dashed arrows in Fig. 6). As before, consider a declaration "$\mathbf{friend}\ View\ \mathbf{reads}\ vsn, val$;" in $Subject3$. We use the terminology *granter* for class $Subject3$ and *friend* for the class $View$ to which access is granted. Here access means that the admissibility condition is relaxed to allow the invariant of class $View$ to depend on vsn and val in $Subject3$. Moreover each instance v of $View$ is required to maintain the following invariant:

If in the current state $\mathcal{I}^{View}[v/\mathbf{self}]$ depends on s then $v \in s.deps$.

One can now adapt the precondition (6) for field update to quantify over just $s.deps$. Special commands $\mathbf{attach}$ and $\mathbf{detach}$ are used to manipulate $deps$, much like $\mathbf{pack}$ and $\mathbf{unpack}$ [7, 40].

The discipline also rectifies another flaw of (6). Instead of requiring that all dependent views are unpacked, we account for the possibility that the update is not going to falsify the invariant of a packed view. For example, suppose that we dropped the requirement, $sbj.vsn - 1 \leq st.vsn$, that a $View$ not lag too far behind, keeping as invariant only this:

$$st.vsn \leq sbj.vsn \wedge (st.vsn = sbj.vsn \Rightarrow st.val = sbj.val) \tag{7}$$

Then an update of the form $vsn, val := vsn + 1, \ldots$ never falsifies the invariant of a view.

More generally, we allow $View$ to declare conditions —visible to the granting class $Subject3$— under which its invariant is not falsified. The declaration

$$\mathbf{guard}\ sbj.vsn := \alpha\ \mathbf{by}\ U \quad \text{where } U =_{df} \alpha - 1 \leq \mathbf{self}.st.vsn \leq \alpha$$

protects the original invariant $\mathcal{I}^{View}$ including condition $sbj.vsn - 1 \leq st.vsn$. This is because the proof obligation imposed on $View$ for U is satisfied:

$$\mathcal{I}^{View} \wedge U \Rightarrow wp(\mathbf{self}.sbj.vsn := \alpha)(\mathcal{I}^{View})$$

Owing to this we can now weaken the precondition (6) for field update, since under condition U the invariant of a packed view cannot be falsified. For update $vsn := vsn + 1$ in code of $Subject3$, the precondition is

$$inv > Subject3 \wedge (\forall\, v \mid v \textbf{ in } deps \;\Rightarrow\; v.inv > View \vee U[\textbf{self}/sbj, v/\textbf{self}, (vsn+1)/\alpha]\,) \tag{8}$$

Just as, for any object o, the field $o.inv$ serves as a publicly visible abstraction of $\mathcal{I}[o/\textbf{self}]$, here U serves to abstract from $wp(\textbf{self}.sbj.vsn := \alpha)(\mathcal{I}^{View})$ in a way suitable to be visible in $Subject$, without fully revealing $\mathcal{I}^{View}$. The substitutions adapt the update guard from the nomenclature of $View$ to that of $Subject3$ and to the particular update $vsn := vsn + 1$.

History Constraints. For the invariant (7), an alternative to the friendship discipline is to use history constraints [30]. A history constraint is a two-state predicate on an object, interpreted as a constraint on any two successive visible states of the object (e.g., states at method call or return). Let us use primes on field names to designate the "after" state, to give an example history constraint that is satisfied by $Subject3$:

$$vsn \leq vsn'$$

That is, vsn increases monotonically. Invariant (7) cannot be falsified by any update of vsn that satisfies the constraint.

In general, if the granter declares a history constraint and the friend's invariant is not falsifiable by updates satisfying the constraint then no precondition concerning the friend needs to be imposed on updates by the granter. To the author's knowledge, history constraints have only been studied in the case where they depend on the object's own fields, not on fields of reps [30]. It could be valuable to study constraints that depend on fields of reps (just as our example update guard depends on fields of the Cache of $View$).

A shortcoming of history constraints is that their meaning depends on a notion of visible state, just like the visible state semantics of invariants. The inv/own discipline dodges this by using field inv to maintain program invariants which are true "at every semicolon". Perhaps there is a comparable notion of history constraint.

It is not clear how to use a history constraint if $\mathcal{I}^{View}$ includes the condition $sbj.vsn - 1 \leq st.vsn$ which we use to force notifications. It is true that the vsn field of a Subject is incremented by one in each atomic update, but the strongest history constraint is that it is nondecreasing, since at some computation steps it is unchanged and after sufficiently many steps it can change by more than one.

An advantage of history constraints is that they handle a sequence of multiple updates to $Subject$ whereas the update guard is formulated in terms of an atomic update.

Update/Yielding. How can a granter establish the U case in precondition (8)? To reason that a given view satisfies U, in the context of $Subject$, it might be possible to use specifications of methods of $View$. In particular, U could be given as postcondition of $notify$.

A history constraint is something like a pre/post specification that applies not to a particular method or command but to arbitrary pairs of observations. One can see an

update guard as a precondition for arbitrary steps; what about a postcondition thereof (in addition to the invariant)? Under precondition U, increment of $sbj.vsn$ by one yields a state where the view's version lags exactly one step behind. This can be declared as a postcondition in the **guard** declaration; the idea is worked out in [7].

8 Challenges for Future Work

We have not exhausted the issues brought up by the last example. The guard U depends on owned objects (the $Cache$) of View, exposing some of the internal state of a view to its $Subject3$. Moreover the $Listener$ could well update the cache, indeed one could imagine that $Listener$ maintains an invariant similar to $\mathcal{I}^{View}$. In Fig. 6 we draw common ownership arrows from the cache to hint that, as in the case of a Collection/Iterators, the situation seems to be one where multiple objects comprise the public interface for an abstraction and have shared access to the reps.

Such increasingly elaborate patterns have motivated increasingly complicated ownership type systems and may well necessitate more complicated versions of the inv/own and friendship disciplines. We mention two more ways in which the friendship discipline, as currently formulated [7, 40], is inadequate. Consider a variation on $Subject3$ where its state is not just val : **int** but rather some data structure; then $\mathcal{I}^{View}$ would depend not on $sbj.val$, but on $sbj.f.g\ldots$, i.e., a path into that data structure. With ownership, the admissibility condition requires that each of sbj, $sbj.f$, $sbj.f.g$ etc. is owned. Friendship instead imposes mutual obligations and it appears nontrivial to generalize the conditions to longer paths owing to the various possibilities of sharing.

The second inadequacy of the current friendship discipline is that each atomic update must restore the friend's invariant. In the case at hand, the friend's invariant depends on two fields val and vsn. It happens that if $sbj.val$ is updated before $sbj.vsn$ then the discipline can be followed, but in general one would like to require the invariant to hold only after several related updates are done. Perhaps this can be achieved by a protocol with a ghost variable to track whether the friendship dependence is in effect? Would that explicit expression of atomicity lead to very different reasoning than using history constraints?

While incremental extensions can be made to address these two inadequacies of the friendship discipline, what really seems to be needed is a general setup for such disciplines. While ownership is widely applicable and provides a strong form of encapsulation at fairly low cost, attempts to extend it to multiple owners or cooperating peers seem more specialized. For example, friendship caters to the situation where one instance of the granter class is depended on by multiple instances of a single other class, the friend. What about situations where several objects of the same class, or of several different classes, are interdependent and collectively provide some abstraction?

Packages are not the answer because a package is a collection of classes and does nothing to describe the configurations in which instances are intended to be deployed. What we seek is a notation in which a design and reasoning pattern can be expressed. A design pattern typically involves a configuration of instances (e.g., subject and view, collection and iterators) with certain operations and protocol. A pattern-specific discipline for reasoning could be based on a single invariant for the pattern's object con-

figuration, expressed with the help of ghost fields to encode the configuration[10] and its protocol; or perhaps the invariant can be decentralized into interdependent object invariants.

Pattern-specific rules would need to be given —stipulated annotations for critical operations including updates of ghost variables to track the structure of interest. Verification of the pattern would involve establishing designated program invariants as a consequence of the stipulated annotations.

About the friendship discipline, Tony Hoare asked "Would it not be better to define a general facility for the user to introduce ghost variables and assertions, rather like aspects in aspect-oriented programming?"[11] Another possible source of inspiration is Separation Logic, which offers notation that can transparently depict groups of objects and their interrelation. In separation logic, quantification over predicates is needed for interesting specifications, in part because patterns of heap structure are expressed using separation at the level of predicates. Why not *expressions* describing regions? Pattern matching for such expressions has been given a semantic foundation [37, 38] but not thoroughly investigated. A less speculative question to be investigated concerns the requirement, in Separation Logic, that invariants be *precise* predicates, i.e., supported by a definite region of the heap [41]. In simple cases, invariants are precise in virtue of being formulated by reachability in some data structure. Ghost structure may offer a scalable and precise shadow of encapsulation.

Acknowledgements. Thanks to the organizers of FMCO 2004 for the opportunity to present this work and meet with other researchers in such a congenial and supportive environment. This paper reflects feedback from the meeting as well as corrections and suggestions from reviewers and from Mike Barnett.

References

[1] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 1–25, 2004.

[2] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 2002. Accepted, revision pending. Extended version of [3].

[3] A. Banerjee and D. A. Naumann. Representation independence, confinement and access control. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, pages 166–177, 2002.

[4] A. Banerjee and D. A. Naumann. State based ownership, reentrance, and encapsulation. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005. To appear.

[5] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS post-proceedings*, 2004.

[7] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In D. Kozen, editor, *Mathematics of Program Construction*, pages 54–84, 2004.

[10] The Aldrich-Chambers system might help here [1].

[11] Personal communication, April 2004.

[8] G. Bierman and M. Parkinson. Separation logic and abstraction. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, pages 247–258, 2005.
[9] L. Birkedal and N. Torp-Smith. Higher order separation logic and abstraction. Submitted., Feb. 2005.
[10] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic reasoning for object-oriented programming. *Sci. Comput. Programming*, 52(1-3):53–100, 2004.
[11] P. H. M. Borba, A. C. A. Sampaio, and M. L. Cornélio. A refinement algebra for object-oriented programming. In L. Cardelli, editor, *European Conference on Object-oriented Programming (ECOOP)*, number 2743 in LNCS, pages 457–482, 2003.
[12] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, 2002.
[13] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, pages 213–223, 2003.
[14] C. Calcagno, P. O'Hearn, and R. Bornat. Program logic and equivalence in the presence of garbage collection. *Theoretical Comput. Sci.*, 298(3):557–581, 2003.
[15] A. L. C. Cavalcanti and D. A. Naumann. Forward simulation for data refinement of classes. In L. Eriksson and P. A. Lindsay, editors, *Formal Methods Europe*, volume 2391 of *LNCS*, pages 471–490, 2002.
[16] D. Clarke. Object ownership and containment. Dissertation, Computer Science and Engineering, University of New South Wales, Australia, 2001.
[17] D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In J. L. Knudsen, editor, *ECOOP 2001 - Object Oriented Programming*, 2001.
[18] F. de Boer and C. Pierik. Computer-aided specification and verification of annotated object-oriented programs. In B. Jacobs and A. Rensink, editors, *Formal Methods for Open Object-Based Distributed Systems*, pages 163–177, 2002.
[19] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
[20] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conf. on Program. Lang. Design and Implementation (PLDI)*, pages 59–69, 2001.
[21] D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Research 156, DEC Systems Research Center, 1998.
[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
[23] C. A. R. Hoare. Proofs of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.
[24] B. Jacobs, J. Kiniry, and M. Warnier. Java program verification challenges. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects (FMCO 2002)*, LNCS.
[25] B. Jacobs, K. R. M. Leino, and W. Schulte. Multithreaded object-oriented programs with invariants. In *SAVCBS*, 2004.
[26] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *LNCS*, pages 262–284. 2003.
[27] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 491–516, 2004.
[28] K. R. M. Leino and P. Müller. Modular verification of static class invariants. In *Formal Methods*, 2005.
[29] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.
[30] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6), 1994.

[31] N. Lynch and F. Vaandrager. Forward and backward simulations part I: Untimed systems. *Information and Computation*, 121(2), 1995.

[32] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, second edition, 1997.

[33] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of Second Intl. Joint Conf. on Artificial Intelligence*, pages 481–489, 1971.

[34] J. C. Mitchell. Representation independence and data abstraction. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, pages 263–276, 1986.

[35] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[36] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. Technical Report 424, Department of Computer Science, ETH Zurich, 2004.

[37] D. A. Naumann. Ideal models for pointwise relational and state-free imperative programming. In H. Sondergaard, editor, *ACM International Conference on Principles and Practice of Declarative Programming*, pages 4–15, 2001.

[38] D. A. Naumann. Patterns and lax lambda laws for relational and imperative programming. Technical Report 2001-2, Computer Science, Stevens Institute of Technology, 2001.

[39] D. A. Naumann. Soundness of data refinement for a higher order imperative language. *Theoretical Comput. Sci.*, 278(1–2):271–301, 2002.

[40] D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract). In *IEEE Symp. on Logic in Computer Science (LICS)*, pages 313–323, 2004.

[41] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, pages 268–280, 2004.

[42] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *Journal of the ACM*, 42(3):658–709, 1995.

[43] C. Pierik, D. Clarke, and F. S. de Boer. Controlling object allocation using creation guards. In *Formal Methods 2005*, 2005.

[44] C. Pierik and F. de Boer. On behavioral subtyping and completeness. In *ECOOP Workshop on Formal Techniques for Java-like Programs*. 2005. To appear.

[45] C. Pierik and F. S. de Boer. A proof outline logic for object-oriented programming. *Theoretical Comput. Sci.*, 2005. to appear.

[46] A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In P. W. O'Hearn and R. D. Tennent, editors, *Algol-Like Languages*, volume 2, chapter 17, pages 173–193. Birkhauser, 1997. Reprinted from *Proceedings Eleventh Annual IEEE Symposium on Logic in Computer Science*, Brunswick, NJ, July 1996, pp 152–163.

[47] A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

[48] G. Plotkin. Lambda definability and logical relations. Technical Report SAI-RM-4, University of Edinburgh, School of Artificial Intelligence, 1973.

[49] Rehof and Mogensen. Tractable constraints in finite semilattices. *Sci. Comput. Programming*, 1996.

[50] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[51] C. Skalka and S. Smith. Static use-based object confinement. *Springer International Journal of Information Security*, 4(1-2), 2005.

[52] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, NY, second edition, 2002.

A Dynamic Binding Strategy for Multiple Inheritance and Asynchronously Communicating Objects

Einar Broch Johnsen and Olaf Owe

Department of Informatics, University of Oslo, Norway
{einarj, olaf}@ifi.uio.no

Abstract. This paper considers an integration of asynchronous communication, virtual binding, and multiple inheritance. Object orientation is the leading paradigm for concurrent and distributed systems, but the tightly synchronized RPC communication model seems unsatisfactory in the distributed setting. Asynchronous messages are better suited, but lack the structure and discipline of traditional object-oriented methods. The integration of messages in the object-oriented paradigm is unsettled, especially with respect to inheritance and redefinition.

Asynchronous method calls have been proposed in the Creol language, reducing the cost of waiting for replies in the distributed environment while avoiding low-level synchronization constructs such as explicit signaling. A lack of reply to a method call need not lead to deadlock in the calling object. Creol has an operational semantics defined in rewriting logic. This paper considers a formal operational model of multiple inheritance, virtual binding, and asynchronous communication between concurrent objects, extending the semantics of Creol.

1 Introduction

Object orientation is the leading paradigm for concurrent and distributed systems. The importance of such systems is increasing in society, driving the need for formal models and reasoning support for object-oriented distributed systems. With the current domination of languages such as Java and C++, one may think that there is only one way to understand object-oriented languages. In the setting of distributed systems, these languages may be criticized for their approach to concurrency as well as to communication. An alternative approach is taken in the Creol language: concurrent objects typed by interfaces which communicate by means of asynchronous method calls. This communication model integrates asynchronous message passing with the high-level structuring mechanism of method definition and invocation [20].

In this paper we discuss multiple inheritance in the setting of open distributed systems and consider the combination of multiple inheritance and virtual (or late) binding. Multiple inheritance provides a flexible way to combine class hierarchies, but is generally considered error-prone. Multiple inheritance is often explained in terms of run-time data structures such as virtual pointer tables [38], which are complex and hard to understand. High-level formal treatments are scarce (e.g., [34,6,3,15]) but needed to clarify intricacies, thus facilitating design and correctness reasoning for programs using multiple inheritance. In this paper, an operational semantics capturing multiple inheritance and virtual binding of methods for Creol is defined, extending work reported in [22].

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 274–295, 2005.
© Springer-Verlag Berlin Heidelberg 2005

In particular, common restrictions on the name space of methods to avoid name conflicts either severely limit the use of class inheritance or become impractical in large class hierarchies and break the encapsulation principle for class inheritance. In order to maintain local reasoning control without abandoning common features of virtual binding such as method overloading and overriding, virtual binding of method calls must be handled carefully. For this purpose, a dynamic *pruned binding strategy* is introduced in this paper and formalized with an operational semantics given in rewriting logic. Rewriting logic [29] is chosen due to its high level of abstraction with inherent support for distribution, concurrency, and asynchronous communication, as well as its simulation and model checking facilities through the Maude tool [8,30]. This allows us to focus the formalization on issues related to inheritance and virtual binding. The strategy is integrated in the Creol interpreter in Maude. An example demonstrates that a form of binding anomaly is avoided with this strategy.

Paper Overview. Sect. 2 provides a background discussion on multiple inheritance. Sect. 3 introduces the Creol language. Sect. 4 extends Creol with mechanisms for multiple inheritance and pruned virtual binding, Sect. 5 illustrates the mechanisms, and Sect. 6 defines its operational semantics. Sect. 7 considers related work and Sect. 8 concludes the paper.

2 Inheritance: Reuse of Behavior and Reuse of Code

Inheritance is a powerful feature of object orientation, but its exact role as a structuring mechanism for programs varies between different object-oriented languages. With *single* inheritance, a class is derived from one direct ancestor class, while with *multiple* inheritance there may be several direct ancestors. Inheritance may be understood as a mechanism for sharing and specialization of behavior as well as code. Formal approaches to inheritance tend to favor the first interpretation and understand inheritance in terms of behavioral reuse, obeying the *substitutability* principle: As a subclass is a specialization of a superclass, an object of the subclass may masquerade as an object of the superclass. This interpretation of inheritance has led to an active field of research on behavioral subtyping [2,27,14], identifying conditions for safe substitutability. A *type* describes a collection of objects which share the same externally observable behavior. Subtyping provides a powerful structuring mechanism for defining, specializing, and understanding the external behavior of objects.

A *class* describes a collection of objects which share the same internal structure; i.e., attributes and method definitions. Code inheritance provides an equally powerful mechanism for defining, specializing, and understanding the imperative structure of classes through code reuse and modification. Class extension and method redefinition are convenient both for development and understanding of code. Calling superclass methods in a subclass method enables *reuse in redefined methods*, making the relationship between the method versions explicit. Thus, this facility is clearly superior to cut-and-paste programming with regard to the ease with which existing code may be inspected and understood and it is also clearly superior to inheritance mechanisms which do not distinguish between locally defined and inherited definitions. A denotational semantics for code sharing and reuse based on single inheritance is given in [9].

Although many languages identify the subclass and subtype relations, in particular for parameter passing, several authors argue that inheritance relations for code and for behavior should be distinct [10,2,5,37]. From the pragmatic point of view, combining these relations leads to severe restrictions on code reuse which seem unattractive to programmers. From a reasoning perspective, the separation of these relations allows greater expressiveness while providing type safety. In order to solve the conflict between unrestricted code reuse in subclasses, and behavioral subtyping and incremental reasoning control [27,37], we use behavioral interfaces [19,21] to type object variables (i.e., references) and external (remote) calls, and allow multiple inheritance for both interfaces and classes. Interface inheritance is restricted to a form of behavioral subtyping, whereas class inheritance may be used freely for code reuse. A class may implement several interfaces, provided that it satisfies their syntactic and semantic requirements. An object of class C supports an interface I if the class C implements I. Reasoning control is ensured by substitutability at the level of interfaces: *an object supporting an interface I may be replaced by another object supporting I or a subinterface of I* in a context depending on I, although the latter object may be of another class. Subclassing is unrestricted in the sense that implementation claims and class invariants are not in general inherited.

With distinct inheritance and subtyping hierarchies, class inheritance could allow a subset of the attributes and methods of a class to be inherited. However, this would require considerable work establishing invariants for parts of the superclass that appear desirable for inheritance, either anticipating future needs or while designing subclasses. The *encapsulation principle* for class inheritance states that it should suffice to work at the subclass level to ensure that the subclass is well-behaved when inheriting from a superclass: Code design as well as new proof obligations should occur in the subclass only. Situations that break this principle are called inheritance anomalies [28,32]. Reasoning considerations therefore suggest that all attributes and methods of a superclass are inherited, but method redefinition may violate the semantic requirements of an interface of the superclass.

2.1 Multiple Inheritance

The focus of this paper is the formalization of an operational semantics for code reuse through class level multiple inheritance. Multiple inheritance seems desirable because it provides much better possibilities for sharing than single inheritance, allowing named features (attributes and methods) from several classes to be integrated. The combination of single inheritance and interfaces is sometimes proposed as an alternative to multiple inheritance, but this approach has some difficulties. In particular virtual binding does not integrate directly with delegation, and the use of private methods as well as program variables from superclasses is problematic. Tempero and Biddle show how the reusability of the Java Core API is adversely affected by the lack of multiple inheritance [39].

Multiple inheritance is found in languages such as C++ [38], CLOS [12], Eiffel [31], Full Maude [8], POOL [2], and Self [7]. Although multiple inheritance provides a flexible way to describe class hierarchies, it is avoided or only allowed in a restricted version (such as interfaces, abstract classes, or traits) in many languages, e.g., Java and C#. Apart from semantic issues, two important arguments against multiple inheritance are: (1) the run-time system of languages with multiple inheritance is more complex and

less efficient, and (2) inheriting from many classes increases the possibility of programmer mistakes. However, efficient run-time systems for languages with multiple inheritance have been developed [25,38]. In order to address argument (2), formal methods may contribute to a better understanding of existing multiple inheritance mechanisms and hopefully contribute to mechanisms with better support for reasoning. The multiple inheritance relation is transitive and defines a class hierarchy structured as a directed acyclic graph. A class in the class hierarchy extends the features declared in its inherited classes (or superclasses), possibly overloading or redefining some of these features. We shall say that a feature is defined *above* a class *C* if it is defined in *C* or in at least one of the classes inherited by *C*.

2.2 Naming Policies for Conflict Resolution

From a reasoning perspective, the difficulties regarding multiple inheritance occur when the name spaces of several inherited classes conflict, both with regard to program variables and methods [24]. A name conflict is *vertical* if a name occurs in a class and in one of its ancestors, corresponding to overridden method declarations. A name conflict is *horizontal* if the name occurs in distinct branches of the graph. While vertical name conflicts are fairly well understood, different solutions have been proposed to deal with horizontal name conflicts. One approach is to remove ambiguities by *explicit resolution*. This is achieved if a name which is inherited from several superclasses is redefined in the subclass or directed to a superclass definition through qualification [38]. If a class is multiply inherited, qualification by class path leads to a duplication of attributes while qualification by class name leads to unification (virtual classes in C++). This choice leads to one or two copies of the attributes of class *A* in Fig. 1b. Path qualification is not always sufficient to distinguish two inherited instances of a class and may therefore fail, as illustrated by Fig. 1c. Explicit resolution can also be achieved by renaming attributes and methods [2,31], thus eliminating name conflicts. When there are no name conflicts, the inheritance graph may be *linearized* and the need for explicit support for multiple inheritance in the run-time system is avoided. This is also the case with mixin-based inheritance [4], and with traits [35]. Mixins and traits are integrated in the linear inheritance graph to extend and modify the resulting behavior of the superclass. However linearization has been criticized for changing the parent-child relationship between classes in the inheritance hierarchy [36].

Ambiguities may also be seen as a natural feature of multiple inheritance, occurring when related methods in different superclass hierarchies are given the same name. From this point of view it seems less desirable to apply a naming discipline which forces the programmer to modify names a posteriori, making the class definitions more difficult to understand. Taking this approach, ambiguities are addressed by *implicit resolution*. Three approaches can be used to explain implicit resolution of ambiguous method names. First, methods with the same name may be seen as equally appropriate. In this case the method definitions may nondeterministically compete for selection, as in Full Maude [8] and the Join-calculus [15]. (Redefinition is not supported by Full Maude, whereas renaming is required in the Join-calculus.) Second, methods of the same name may be jointly selected, extending the binding strategy of Beta to multiple inheritance. Third, ambiguities may be solved by fixing the *order* of the inherited classes; this way

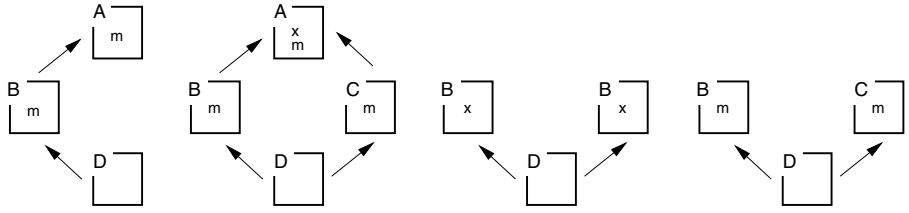


Fig. 1. Examples of class inheritance: (a) single inheritance, (b) a common ancestor in the inheritance graph, (c) duplicate inheritance, and (d) multiple inheritance

the strategy for selecting method definitions will be unambiguous [12,22,7]. This seems desirable as it leaves the programmer in control.

2.3 Virtual Binding

Virtual binding (or dynamic dispatch) is a powerful mechanism of object orientation, originally introduced in [11] for single inheritance in Simula. A method is virtually bound if the body corresponding to a method invocation is selected at run-time. Virtual binding is applied when the actual class of an object is not statically known. Traditionally, this happens when a method invoked from a class is overridden above the actual class of the object. When objects are typed by interfaces, many classes may implement the same interface. Consequently all external method calls are virtually bound.

Combined with class inheritance, virtual binding allows programming with the so-called template method pattern [16]: a base class provides architecture and subclasses provide the specialized (auxiliary) methods, while code reuse is supported for the architecture. The mechanism can be illustrated by an object of class *D* which executes a method defined in its superclass *A* (cf. Fig. 1a) and this method makes a call to a method *m*. With virtual binding, the code selected for execution will be associated to the first matching signature for *m above D*; i.e., the method in *B* is selected. However it is unsettled how to virtually bind method invocations in a class hierarchy with multiple inheritance, if methods are defined in different classes in the hierarchy. In the example of Fig. 1b, a strategy is needed to clearly express which method definition to select among the candidates for *m*.

Formal models of possible solutions to multiple inheritance may contribute to better understanding and use of multiple inheritance, and facilitate reasoning about code inheritance. A denotational account of multiple inheritance has been given [6], but virtual binding is not considered as name conflicts are assumed not to occur.

An operational semantics in rewriting logic allows executable experimentation with different strategies for virtual binding. For this purpose, we consider multiple inheritance in the setting of the Creol language, which has a complete formalization in rewriting logic. In previous work [22], an ordered solution was proposed in which the binding strategy did not distinguish a virtual call from a superclass (*A* in Fig. 1a) and a standard call from the subclass (*C* in Fig. 1a). In this paper a novel version of the ordered approach is considered in which the order may vary between calls, as the ordering is dynamically decided by the context of each call. This new approach, called *pruned*

binding, avoids renaming while providing better support for the encapsulation principle. Calls are always bound to specializations of the definition found by static analysis, allowing reasoning reuse for virtual calls in the setting of multiple inheritance.

Consider the case where *D* inherits two unrelated classes *B* and *C* (Fig. 1d), both with a method *m*. Assume that a *D* object calls a method in *C* which in turn calls *m* locally. With the ordered approach this call will bind to the *m* of *B* rather than that of *C*, assuming no redefinition of *m* in *D*. This binding is clearly undesirable since the *m* of *B* is not a redefinition of that of *C*. The two *m* methods have no relationship since they are from unrelated class hierarchies. The example in Sect. 5 demonstrates resulting problems. These problems are avoided with the pruned binding strategy. Furthermore, the strategy ensures the principle that when the actual class of an object is smaller, each local call will be bound to a smaller class. This principle is intuitive and is also useful for reasoning control.

3 A Language for Asynchronously Communicating Objects

This section provides a basis for the technical discussion which follows. We consider a small object-oriented language which is a subset of Creol [20,22], a high-level language for distributed concurrent objects. We distinguish data, typed by data types, and objects, typed by interfaces. The language allows both blocking and nonblocking method calls, based on a uniform semantics. Attributes (instance variables) and method declarations are organized in classes, which may have data and object parameters. Objects are concurrent and have their own processor which evaluates local processes. A process consists of program code with *processor release points* together with a local state, representing remaining parts of method activations. Processes may be *active*, reflecting autonomous behavior initiated at creation time by the *run* method, or *reactive*; i.e., in response to method invocations. Due to processor release points, the evaluation of processes may be interleaved. The values of an object's program variables may depend on the nondeterministic interleaving of processes. However, a method activation may have local variables supplementing the object variables, in particular the values of formal parameters are stored locally. An object may contain several (pending) activations of the same method, possibly with different values for local variables.

Guards *b* in statements **await** *b* explicitly declare potential processor release points. When a guard which evaluates to false is encountered during process evaluation, the process is *suspended* and the processor released. After processor release, any enabled pending process may be selected for evaluation. For the examples of this paper, it suffices to consider guards as boolean expressions over program variables, but we introduce reply guards in the operational semantics (cf. Sect. 6.5).

Statements can be composed sequentially or by conditional branching. Let S_1 and S_2 denote statement lists. Sequential composition may introduce inner guards: **await** *b* is a potential release point in S_1;**await** *b*;S_2. Assignment to local and object variables is expressed as V : E for a disjoint list of program variables V and an expression list E, of matching types. The reserved word *self* is used for self reference. In-parameters as well as the *self* and *caller* pseudo-variables are read-only variables.

All object interaction happens through method calls. We consider here blocking calls and nonblocking calls. (The full language provides more expressiveness [20].)

A *nonblocking method call* is written **await** $x.m(\mathrm{E};\mathrm{V})$. The calling process emits the call to an object x and suspends itself while waiting for a reply. When the reply arrives, return values are assigned to V and evaluation continues.

A *blocking method call*, immediately blocking the processor while waiting for a reply, is written $x.m(\mathrm{E};\mathrm{V})$. When x evaluates to *self*, the call is said to be local. The language does not support monitor reentrance (except for calls to self), mutual blocking calls may therefore lead to deadlock. In order to evaluate local blocking calls, the evaluation of the call will precede the continuation of the active process, thereby unblocking the processor (self-reentrance).

Internal calls are not prefixed by an object identifier and are identified syntactically, otherwise the call is external. All calls are virtually bound, except when the method name is explicitly qualified by a class name, $m@C$. In our setting method calls can always be emitted, as a receiving object cannot block communication. *Method overtaking* is allowed: if methods offered by an object are invoked in one order, the object may start execution of the method activations in another order.

With nonblocking method calls, the object will not block while waiting for replies. This approach allows flexibility in the distributed setting: suspended processes or new method calls may be evaluated while waiting. If the called object never replies, deadlock is avoided as other activity in the object is possible. However, when the reply arrives, the *continuation* of the process must compete with other pending and enabled processes.

4 Multiple Inheritance

A mechanism for multiple inheritance is now considered, where all attributes and methods of a superclass are inherited by the subclass, and where superclass methods may be redefined. In the syntax, the keyword **inherits** is introduced followed by an *inheritance list*; i.e., a list of class names $C(\mathrm{E})$ where E provides the actual class parameters.

Let a class hierarchy be a directed acyclic graph of parameterized classes. Each class consists of a list of inherited classes, a set of attributes (program variables including class parameters), and method definitions. The encapsulation provided by interfaces suggests that external calls to an object of class C are virtually bound to the closest method definition above C. However, the object may internally invoke methods of its superclasses. In the setting of multiple inheritance and overloading, methods defined in a superclass may be accessed from the subclass by qualified references. Vertical name conflicts for method names are resolved in a standard way: the first matching definition with respect to the types of the actual parameters is chosen while ascending a branch of the inheritance tree. Horizontal name conflicts will be resolved dynamically depending on the class of the object and the context of the call.

4.1 Qualified Names

Qualified names may be used to uniquely refer to an attribute or method in a class. For this purpose, we adapt the **qua** construct of Simula to the setting of multiple inheritance. For an attribute x or a method m declared in a class C, we denote by $x@C$ and $m@C$ the qualified names which provide static references to x and m. By extension, if x or m

Syntactic categories.		*Definitions.*
s in Stm	v in Var	$p ::= m \mid x.m \mid m@classname \mid m < classname$
t in Label	e in Expr	$\mathrm{S} ::= s \mid s; \mathrm{S}$
m in Mtd	x in ObjExpr	$s ::= \mathbf{skip} \mid (\mathrm{S}) \mid \mathrm{V} := \mathrm{E} \mid v := \mathbf{new}\ classname(\mathrm{E})$
p in MtdCall	b in BoolExpr	$\mid p(\mathrm{E};\mathrm{V}) \mid \mathbf{await}\ p(\mathrm{E};\mathrm{V}) \mid \mathbf{await}\ b \mid \mathbf{if}\ b\ \mathbf{then}\ \mathrm{S}_1\ \mathbf{else}\ \mathrm{S}_2\ \mathbf{fi}$

Fig. 2. A subset of the Creol language for method definitions, with typical terms for each category. Capitalized terms such as E denote lists, sets, or multisets of the given syntactic categories.

is *not* declared in C, but inherited from the superclasses of C, the qualified reference $m@C$ binds as an unqualified reference m from C.

Attribute names are not visible through an object's external interfaces. Consequently attribute names should not be merged if inheritance leads to name conflicts, and attributes of the same name should be allowed in different classes of the inheritance hierarchy [36]. In order to allow the reuse of attribute names, these will always be expanded into qualified names. This is desirable in order to avoid run-time errors that may occur if methods of superclasses assign to overloaded attributes. This qualification convention has the following consequence: unlike C++, there is no duplication of attributes when branches in the inheritance graph have a common superclass. Consequently if multiple copies of the superclass attributes are needed, one has to rely on delegation techniques.

Instantiation of Attributes. At object creation time, attributes are collected from the object's class and superclasses. An attribute in a class C is declared by **var** $x : T = e$, where x is the name of the attribute, T its type, and e its initial value. The expression e may refer to the values of the class parameter variables V, as well as to the values of inherited attributes by means of qualified references. The initial state values of an object of class C then depend on the actual parameter values bound to V. These may be passed as parameter values to inherited classes in order to derive values for the inherited attributes, which in turn may be used to instantiate the locally declared attributes.

Accessing Inherited Attributes and Methods. If C is a superclass of C', we introduce the syntax **await** $m@C(\mathrm{E};\mathrm{V})$ for nonblocking, and $m@C(\mathrm{E};\mathrm{V})$ for blocking, internal calls to a method above C in the inheritance graph. These calls may be bound without knowing the exact class of the *self* object, so they are called *static*, in contrast to calls without @, called *virtual*. We assume that attributes have unique names in the inheritance graph; this may be enforced at compile time by extending each attribute name x with the name of the class in which it is declared, which implies that attributes are bound statically. Consequently, a method declared in a class C may only access attributes declared above C. In a subclass, an attribute x of a superclass C is accessed by the qualified reference $x@C$. This means that multiply inherited superclasses are shared, rather than duplicated. Duplication may be achieved by class renaming in an inheritance list. The language syntax is summarized in Fig. 2.

4.2 Virtual Binding

Let the nominal subtype relation $\prec$ be a reflexive partial ordering on types, including interfaces. A data type may only be a subtype of a data type and an interface only of an

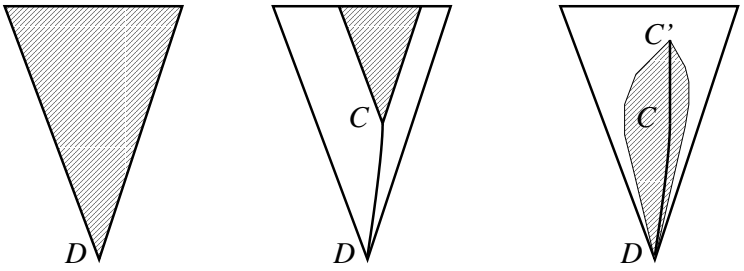


Fig. 3. Binding calls to m, $m@C$, and $m < C'$ from class D

interface. If $T \prec T'$ then any value of T may masquerade as a value of T'. For product types R and R', $R \prec R'$ is the point-wise extension of the subtype relation; i.e., R and R' have the same length l and $T_i \prec T_i'$ for every i ($0 \leq i \leq l$) and types T_i and T_i' in position i in R and R' respectively. To explain the typing and binding of methods, $\prec$ is extended to function spaces $A \to B$, where A and B are (possibly empty) product types:

$$A \to B \prec A' \to B' = A \prec A' \wedge B' \prec B$$

expressing the relationship between actual and formal parameters, but not subtyping over function spaces, which are not part of the functional language. The static analysis of an internal call $m(\mathrm{E};\mathrm{V})$ will assign unique types to the in and out parameter depending on the textual context, say that the parameters are textually declared as $\mathrm{E} : T_\mathrm{E}$ and $\mathrm{V} : T_\mathrm{V}$. The call is *type correct* if there is a method declaration $m : A \to B$ in the class C, possibly inherited, such that $T_\mathrm{E} \to T_\mathrm{V} \prec A \to B$. The binding of an internal nonblocking call **await** $m(\mathrm{E};\mathrm{V})$ is handled as the corresponding blocking call $m(\mathrm{E};\mathrm{V})$. An external call to an object of interface I is type correct if it can be bound to a method declaration in I in a similar way. The static analysis of a class will verify that it implements the methods declared in its interfaces.

Let a class C be *below* a class C' if C is C', or is a direct or indirect subclass of C'. Similarly, a method declaration inside a class C is *below* a class C' if C is below C'. We introduce the syntax $m < C'$ for *constrained method calls*, restricting the virtual binding of m to methods below C'. (Static typing requires the class enclosing the call to be below C'.) The pruned virtual binding of method calls is now explained. (The formalization is given in Sec. 6.4.) At run-time, a call to a method of an object o will always be bound above the class of o. Let m be a method declared in an interface I and let o be an instance of a class C implementing I. There are two cases:

1. m is called externally, in which case C is not statically known. In this case, C is dynamically identified as the class of o.
2. m is called internally from C', a superclass of the actual class C of o. In this case static analysis will identify the call with a declaration of m above C', say in C''. Consequently, we let the call be constrained by C'', and compilation replaces the reference to m with a reference to $m < C''$.

The dynamically decided context of a call may eliminate parts of the inheritance graph above the actual class of the callee with respect to the binding of a specific call. If a

method name is ambiguous within the dynamic constraint, we assume that any solution is acceptable. For a natural and simple model of priority, the call will be bound to the first matching method definition above C, in a left-first depth-first order. (An arbitrary order may be obtained by replacing the inheritance list by a multiset.)

It is easy to see that run-time binding always succeeds in any well-typed program. When a method $m : T_E \to T_V$ in an object o of interface I is externally called at run-time, the actual class C of o is dynamically decided and the virtual binding mechanism will bind to a declaration $m : A \to B$ such that $T_E \to T_V \prec A \to B$, taking the first such m when traversing the inheritance graph above C. Static analysis guarantees that C implements I and consequently that at least one method declaration of m above C may be bound to the call. An internal call $m : T_E \to T_V$ is made by an object of a subclass D of C (from the static analysis) and the virtual binding mechanism will bind to a declaration of $m : A' \to B'$ such that $T_E \to T_V \prec A' \to B'$, following the binding strategy constrained by D. Because C is inherited by D, the virtual binding is guaranteed to succeed. However, it is not guaranteed that the declaration above C which was found by static analysis will be selected. In order to ensure that a call to m in D will choose the declaration above C, the method may be qualified as $m@C$ in D. For virtual calls from a superclass C' of C, such qualification cannot be used. In order to ensure that a virtually bound call from a superclass will select a specialization of the statically found declaration, the binding will be constrained by C'. Even if no specialization is found, the binding will succeed as the constraint does not remove the declaration found by static analysis.

5 Example: Combining Authorization Policies

In a database containing sensitive information and different authorization policies, the information returned for a request will depend on the clearance level of the agent making the request. Let Any denote the interface of arbitrary objects, Agent the interface of agents, and Auth an authorization interface with methods *grant*(x), *revoke*(x), *auth*(x), and *delay* for agents x. The two classes *SAuth* and *MAuth*, both implementing Auth, implement single and multiple authorization policies, respectively. Since the attribute *gr* in *SAuth* is implemented as an object identifier, *SAuth* only authorizes one agent at a time whereas *MAuth* authorizes multiple agents. The method *grant*(x) returns when x becomes authorized, and authorization is removed by *revoke*(x). The method *auth*(x) suspends until x is authorized, and *delay* returns once no agent is authorized.

```
class SAuth implements Auth
begin with Any
  var gr: Agent= null
  op grant(in x:Agent) == delay; gr := x
  op revoke(in x:Agent) ==
      if gr = x then gr := null fi
  op auth(in x:Agent) == await (gr = x)
  op delay == await (gr = null)
end
```

```
class MAuth implements Auth
begin with Any
  var gr: Set[Agent] = ∅
  op grant(in x:Agent) == gr := gr ∪ {x}
  op revoke(in x:Agent) == gr := gr \ {x}
  op auth(in x:Agent) == await (x ∈ gr)
  op delay == await (gr = ∅)
end
```

Authorization Levels. *Low clearance* agents may share access to unclassified data while *high clearance* agents have unique access to (classified) data. Proper usage is defined by two interfaces, defining open and close operations at both access levels:

```
interface High
begin with Agent
  op openH(out ok:Bool)
  op access(in k:Key; out y:Data)
  op closeH
end
```

```
interface Low
begin with Agent
  op openL
  op access(in k:Key; out y:Data)
  op closeL
end
```

When the *openH* method returns, the calling agent would not know whether high access was granted, unless a boolean out parameter is present.

Let a class *DB* provide the actual operations on the database. We assume given the operations *access*(**in** k:Key, high:Bool; **out** y:Data), where *high* defines the access level, and *clear*(**in** x : Agent; **out** b : Bool) to give clearance to sensitive data for agent x. Any agent may get low access rights, while only agents cleared by the database may be granted exclusive high access. The class *MAuth* will authorize low clearance, and *SAuth* will authorize high clearance. *SAuth* authorizes only one agent at a time.

```
class HAuth implements High
  inherits SAuth, DB
begin with Agent
  op openH(out ok:Bool) ==
       await clear(caller;ok);
       if ok then grant(caller) fi
  op access(in k:Key; out y:Data) ==
       auth(caller);
       await access@DB(k,true; y)
  op closeH == revoke(caller)
end
```

```
class LAuth implements Low
  inherits MAuth, DB
begin with Agent
  op openL == grant(caller)
  op access(in k:Key; out y:Data) ==
       auth(caller);
       await access@DB(k,false; y)
  op closeL == revoke(caller)
end
```

The code given here uses nonblocking calls whenever there is a possibility of local deadlock. Thus, objects of the four classes above will be able to respond to new requests even when used improperly, for instance when agent access is not initiated by open. Notice that the *caller* pseudo-variable is used to pass on agent identity in local calls. The **with** Agent clauses imply that Agent is the type of *caller*, ensuring strong typing.

The database itself has no interface containing *access*, therefore all database access is through the High and Low interfaces. Notice also that objects of the *LAuth* and *HAuth* classes may not be used through the Auth interface. This would have been harmful for the authorization provided in the example. For instance, a call to *grant* to a *HAuth* object could then result in high *access* without clearance of the calling agent! This supports the approach not to inherit implementation clauses.

Combining Authorization Levels. High and low authorization policies may be combined in a subclass *HLAuth* which implements both interfaces, inheriting *LAuth* and *HAuth*.

```
class HLAuth implements High, Low
  inherits LAuth, HAuth
begin with Agent
  op access(in k:Key; out y:Data) == if caller=gr@SAuth
    then access@HAuth(k; y) else access@LAuth(k; y) fi
end
```

Notice that the same database is used for both High and Low interaction. Although the *DB* class is inherited twice, *HLAuth* gets only one copy (cf. Sect. 4.1).

The example demonstrates natural usage of classes and multiple inheritance. Nevertheless, it reveals problems with the combination of inheritance and *statically ordered* virtual binding: Objects of the classes *LAuth* and *HAuth* work well, in the sense that agents opening access through the Low and High interfaces get the appropriate access. However the addition of the common subclass *HLAuth* is detrimental, assuming a fixed inheritance ordering: When used through the High interface, this class would allow *multiple high access* to data! Calls to the High operations of *HLAuth* will trigger calls to the *HAuth* methods. From these methods the virtual internal calls to *grant*, *revoke*, and *auth* will now bind to those of the *MAuth* class, if selected in a left-most depth-first traversal of the inheritance tree of the actual class *HLAuth*. Note that if the inheritance ordering in *HLAuth* were reversed, similar problems occur with the binding of Low interaction.

The *pruned* binding strategy proposed in this paper ensures that the virtual internal calls inside classes *HAuth* and *LAuth* will be bound in classes *SAuth* and *MAuth*, respectively, regardless of the actual class of the caller (*HAuth*, *LAuth*, or *HLAuth*) and of the inheritance ordering in *HLAuth*. In particular the *grant* call inside *HAuth* will be understood as $grant < SAuth$, which may not bind to *grant* of *MAuth*.

6 An Operational Semantics of Inheritance and Virtual Binding

The operational semantics is defined using rewriting logic [29]. A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$, where the signature Σ defines the function symbols of the language, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. Each rewrite rule describes how a part of a configuration can evolve in one transition step. If rewrite rules may be applied to non-overlapping subconfigurations, the transitions may be performed in parallel. Consequently, concurrency is implicit in rewriting logic (RL). A number of concurrency models have been successfully represented in RL [29,8], including Petri nets, CCS, Actors, and Unity, as well as the ODP computational model [33]. RL also offers its own model of object orientation [8].

Informally, a state configuration in RL is a multiset of terms of given types. Types are specified in (membership) equational logic (Σ, E), the functional sublanguage of RL which supports algebraic specification in the OBJ [17] style. When modeling computat-

ional systems, configurations may include the local system states. Different parts of the system are modeled by terms of the different types defined in the equational logic.

RL extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules, supplementing the equations which define the term language. Assuming that all terms can be reduced to normal form, rewrite rules transform terms modulo the defining equations of E. Conditional rewrite rules are allowed, where the condition is formulated as a conjunction of rewrites and equations which must hold for the main rule to apply:

$$\mathit{subconfiguration} \longrightarrow \mathit{subconfiguration}\ \textbf{\textit{if}}\ \mathit{condition}.$$

Rules in RL may be formulated at a high level of abstraction, closely resembling a compositional operational semantics. In fact, structural operational semantics can be uniformly mapped into RL specifications [30].

6.1 System Configurations

A method call will be reflected by a pair of messages, and object activity will be organized around a *message queue* which contains incoming messages and a *process queue* which contains suspended processes, i.e. remaining parts of method activations. Messages have the general form *message* **to** *dest* where *dest* is a single object or class, or a list of classes. A state configuration is a multiset combining Creol objects, classes, and messages. (In order to increase the parallelism in the model, message queues could be external to object bodies as shown in [20,22].) As usual in RL, the associative constructor for lists, as well as the associative and commutative constructor for multisets, are represented by whitespace.

In RL, objects are commonly represented by terms of the type $\langle o : C \,|\, a_1 : v_1, \ldots, a_n : v_n \rangle$ where o is the object's identifier, C is its class, the a_i's are the names of the object's attributes, and the v_i's are the corresponding values [8]. We adopt this form of presentation and define Creol objects and classes as RL objects. Omitting RL types, a Creol object is represented by an RL object $\langle \mathit{Ob} \mid \mathit{Cl}, \mathit{Pr}, \mathit{PrQ}, \mathit{Lvar}, \mathit{Att}, \mathit{Lab}, \mathit{EvQ} \rangle$, where *Ob* is the object identifier, *Cl* the class name, *Pr* the active process code, *PrQ* a multiset of suspended processes with unspecified queue ordering, *EvQ* a multiset of unprocessed messages, and *Lvar* and *Att* the local and object state, respectively. Let τ be a type partially ordered by $<$, with least element 1, and let $\mathit{next} : \tau \to \tau$ be such that $\forall x \,.\, x < \mathit{next}(x)$. *Lab* is the method call identifier corresponding to labels in the language, of type τ. Thus, the object identifier *Ob* and the generated local label value provide a globally unique identifier for each method call.

The classes of Creol are represented by RL objects $\langle \mathit{Cl} \,|\, \mathit{Inh}, \mathit{Att}, \mathit{Mtds}, \mathit{Tok} \rangle$, where *Cl* is the class name, *Inh* is the inheritance list, *Att* a list of attributes, *Mtds* a multiset of methods, and *Tok* is an arbitrary term of type τ. When an object needs a method, it is bound to a definition in the *Mtds* multiset of its class or of a superclass.

In RL's object model [8], classes are not represented explicitly in the system configuration. This leads to ad hoc mechanisms to handle object creation, which we avoid by explicit class representation. The Creol construct *new* $C(\mathrm{E})$ creates a new object with a unique object identifier, attributes as listed in the class parameter list and in *Att*, and places the code from the *run* method in *Pr*.

6.2 Concurrent Transitions

Concurrent change is achieved in the operational semantics by applying concurrent rewrite steps to state configurations. There are four different kinds of rewrite rules:

- *Rules that execute code from the active process:* For every program statement there is at least one rule. For example, the assignment rule for the program V := E binds the values of the expression list E to the list V of local and object variables.
- *Rules for suspension of the active process:* When an active process guard evaluates to false, the process and its local variables are suspended, leaving *Pr* empty.
- *Rules that activate suspended processes:* When *Pr* is empty, suspended processes may be activated. When this happens, the local state is replaced.
- *Transport rules:* These rules move messages into the message queues, representing network flow.

When auxiliary functions are needed in the semantics, these are defined in equational logic, and are evaluated in between the state transitions [29]. The rules related to method calls, virtual binding, and object creation are now considered in detail. In the presentation irrelevant attributes are ignored in the style of Full Maude [8].

6.3 Method Calls

Blocking and nonblocking calls are given a uniform semantics. In the operational semantics, objects communicate by sending messages. Two messages encode a method call. We here assume that the types of the actual in- and out-parameters of the call have been added to the method invocation as an additional argument *Sig* at compile time. If an object o_1 calls a method m of an object o_2, with actual type *Sig* and actual parameters *In*, and the execution of $m(Sig, In)$ results in the return values *Out*, the call is reflected by two messages $invoc(m, Sig, (n\ o_1\ In))$ **to** o_2 and $comp(n, Out)$ **to** o_1, which represent the invocation and completion of the call, respectively. In the asynchronous setting, invocation messages must include the caller's identity, so completions can be transmitted to the correct destination. Objects may have several pending calls to another object, so the completion message includes a locally unique label value n, generated by the caller.

A blocking call $p(Sig, In; \mathrm{V})$, where V is a list of variables and p one of the forms $x.m$, $m@C$, or $m < C$, is translated into an *asynchronous call*, $!p(Sig, In)$, immediately followed by a blocking *reply statement*, $n?(\mathrm{V})$, where n is the label value uniquely identifying the call:

$$\langle o : Ob \mid Pr : p(Sig, In; \mathrm{V}); \mathrm{S}, Lab : n \rangle = \langle o : Ob \mid Pr : !p(Sig, In); n?(\mathrm{V}); \mathrm{S}, Lab : n \rangle$$

A nonblocking call is understood as an asynchronous call followed by a *reply guard*:

$$\begin{aligned}&\langle o : Ob \mid Pr : \textbf{await}\ p(Sig, In; \mathrm{V}); \mathrm{S}, Lab : n \rangle \\ &\quad = \langle o : Ob \mid Pr : !p(Sig, In); \textbf{await}\ n?; n?(\mathrm{V}); \mathrm{S}, Lab : n \rangle\end{aligned}$$

A reply guard **await** $n?$ evaluates to true when a *comp* message with the label value n has arrived, in which case the reply statement $n?(\mathrm{V})$ will assign the return values to V, otherwise the active process is suspended (see below). Consequently, it suffices to

consider asynchronous invocations, and blocking and guarded replies to capture both blocking and nonblocking method calls.

When an object calls an external method, a message is placed in the configuration:

$$\langle o : Ob \mid Pr : !x.m(Sig, In); \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A}, Lab : n \rangle$$
$$\longrightarrow$$
$$\langle o : Ob \mid Pr : \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A}, Lab : next(n) \rangle$$
$$invoc(m, Sig, (n\ o\ eval(In, (\mathrm{A};\mathrm{L})))) \ \mathbf{to}\ eval(x, (\mathrm{A};\mathrm{L}))$$

where x is an object expression, m a method name, and *eval* is a function which evaluates an expression (list) in the context of a state. When x evaluates to o, the object creates an *invoc* message to itself. Similarly, an internal call gives rise to the same invocation message:

$$\langle o : Ob \mid Pr : !p(Sig, In); \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A}, Lab : n \rangle$$
$$\longrightarrow$$
$$\langle o : Ob \mid Pr : \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A}, Lab : next(n) \rangle$$
$$invoc(p, Sig, (n\ o\ eval(In, (\mathrm{A};\mathrm{L})))) \ \mathbf{to}\ o$$

where p is of the form $m@C$ or $m < C$. The constraint C will be used in the virtual binding as described below.

Transport rules take charge of messages, which eventually arrive at the destination's message queue:

$$(invoc(\mathrm{E})\ \mathbf{to}\ o)\ \langle o : Ob \mid EvQ : \mathrm{Q} \rangle \longrightarrow \langle o : Ob \mid EvQ : \mathrm{Q}\ invoc(\mathrm{E}) \rangle$$
$$(comp(\mathrm{E})\ \mathbf{to}\ o)\ \langle o : Ob \mid EvQ : \mathrm{Q} \rangle \longrightarrow \langle o : Ob \mid EvQ : \mathrm{Q}\ comp(\mathrm{E}) \rangle$$

These rules model loose distribution of objects. Message overtaking is captured by the nondeterminism inherent in RL: messages sent by an object to another object in one order may arrive in any order.

The caller may wait for a completion in a reply statement to synchronize on the completion of the call, or in a reply guard. The reply statement $n?(\mathrm{V})$ blocks until the appropriate reply message has arrived in the message queue. This blocking is captured by a rule requiring matching label values in the active statement and the event queue:

$$\langle o : Ob \mid Pr : (n?(\mathrm{V});\mathrm{S}), EvQ : \mathrm{Q}\ comp(n, Out) \rangle$$
$$\longrightarrow \langle o : Ob \mid Pr : (\mathrm{V} := Out; \mathrm{S}), EvQ : \mathrm{Q} \rangle$$

In the model, EvQ is a multiset; thus the rule will match any occurrence of $comp(n, Out)$ in the queue. Remark that blocking reply statements associated with calls to self require special treatment in order to avoid deadlock [20].

6.4 Virtual and Static Binding of Method Calls

In order to allow concurrent and dynamic execution, the full inheritance graph will not be statically given. Rather, the binding mechanism dynamically inspects the current class hierarchy as present in the configuration. Our approach to virtual binding is to use a *bind* message to be sent from a class to its superclasses, resulting in a *bound* message returned to the object generating the *bind* message. This way, the inheritance graph is explored dynamically and as far as necessary when needed. When the invocation of a

method m is found in the message queue of an object o, a message $\mathit{bind}(o,m,\mathit{In},C)$ can be generated by dynamically retrieving the class C of the object. Here Sig is the method signature as provided by the caller and In is the list of actual in-parameters:

$$\langle o : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{EvQ} : \mathit{invoc}(m, \mathit{Sig}, \mathit{In})\ \mathrm{Q}\rangle$$
$$\longrightarrow \langle o : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{EvQ} : \mathrm{Q}\rangle\ (\mathit{bind}(m, \mathit{Sig}, \mathit{In}, o)\ \textbf{to}\ C)$$

The same applies to internal static calls $m@C$. Static method calls are generated without inspecting the *actual* class of the callee, thus surpassing local definitions:

$$\langle o : \mathit{Ob} \mid \mathit{EvQ} : \mathit{invoc}(m@C, \mathit{Sig}, \mathit{In})\ \mathrm{Q}\rangle \longrightarrow \langle o : \mathit{Ob} \mid \mathit{EvQ} : \mathrm{Q}\rangle\ (\mathit{bind}(m, \mathit{Sig}, \mathit{In}, o)\ \textbf{to}\ C)$$

If a suitable m is defined locally in C, a process with the method code and local state is returned in a *bound* message. Otherwise, the *bind* message is retransmitted to the superclasses of C in a left-first, depth-first order. In order to easily traverse the inheritance graph, an inheritance list is used as the destination of the *bind* message:

$$(\mathit{bind}(m, \mathit{Sig}, \mathit{In}, o)\ \textbf{to}\ C\ \mathrm{I})\ \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}\rangle$$
$$\longrightarrow\ \textbf{if}\ \mathit{match}(m, \mathit{Sig}, \mathrm{M})\ \textbf{then}\ \mathit{bound}(\mathit{get}(m, \mathrm{M}, \mathit{In}))\ \textbf{to}\ o$$
$$\textbf{else}\ \mathit{bind}(m, \mathit{Sig}, \mathit{In}, o)\ \textbf{to}\ (\mathrm{I}'\ \mathrm{I})\ \textbf{fi}\ \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}\rangle$$

The auxiliary predicate $\mathit{match}(m, \mathit{Sig}, \mathrm{M})$ evaluates to true if m is declared in M with a signature Sig' such that $\mathit{Sig} \prec \mathit{Sig}'$, and the function *get* returns a process with the method's code and local state from the method multiset M of the class. (Static checking ensures that virtual binding will succeed.) Values of the actual in-parameters In, the caller o', and the label value n are stored locally. The process w resulting from the binding is loaded into the internal process queue:

$$(\mathit{bound}(w)\ \textbf{to}\ o)\ \langle o : \mathit{Ob} \mid \mathit{PrQ} : \mathrm{W}\rangle \longrightarrow \langle o : \mathit{Ob} \mid \mathit{PrQ} : \mathrm{W}\ w\rangle$$

Note that the use of rewrite rules rather than equations mimics distributed and concurrent processing of method lookup.

Internal Virtual Binding. The binding of an internal virtual call $m < C'$ constrained by C' is slightly more complex. When a match in a class C is found, the inheritance graph of C is checked to ensure that C is below C', otherwise the binding must resume:

$$(\mathit{bind}(m < C', \mathit{Sig}, \mathit{In}, o)\ \textbf{to}\ C\ \mathrm{I})\ \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}\rangle$$
$$\longrightarrow\ \textbf{if}\ \mathit{match}(m, \mathit{Sig}, \mathrm{M})\ \textbf{then}\ (\mathit{find}(C', C)\ \textbf{to}\ C)\ (\mathit{stopbind}(m < C', \mathit{Sig}, \mathit{In}, o)\ \textbf{to}\ C\ \mathrm{I})$$
$$\textbf{else}\ \mathit{bind}(m, \mathit{Sig}, \mathit{In}, o)\ \textbf{to}\ (\mathrm{I}'\ \mathrm{I})\ \textbf{fi}\ \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}\rangle$$

$$(\mathit{found}(b, C')\ \textbf{to}\ C)\ (\mathit{stopbind}(m, \mathit{Sig}, \mathit{In}, o)\ \textbf{to}\ C\ \mathrm{I})\ \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}\rangle$$
$$\longrightarrow\ \textbf{if}\ b\ \textbf{then}\ \mathit{bound}(\mathit{get}(m, \mathrm{M}, \mathit{In}))\ \textbf{to}\ o\ \textbf{else}\ \mathit{bind}(m, \mathit{Sig}, \mathit{In}, o)\ \textbf{to}\ \mathrm{I}\ \textbf{fi}$$
$$\langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}\rangle$$

where *stopbind* is an additional message used to suspend binding while checking that C is below C'. This is done by two auxiliary messages: The message $\mathit{find}(C, o)$ **to** I represents that o is asking I if C is found in I or further up in the hierarchy, whereas $\mathit{found}(b, C)$ **to** o gives the answer to o, where the boolean b is true if the request was successful. This can be formalized by the rewrite rules (ignoring class parameter lists)

$\mathit{find}(C,o)$ **to** $\varepsilon \longrightarrow \mathit{found}(\mathit{false},C)$ **to** o

$\mathit{find}(C,o)$ **to** $\mathrm{I}\ C\ \mathrm{I}' \longrightarrow \mathit{found}(\mathit{true},C)$ **to** o

$(\mathit{find}(C,o)$ **to** $C'\ \mathrm{I})\ \langle C' : Cl \,|\, Inh : \mathrm{I}' \rangle \longrightarrow (\mathit{find}(C,o)$ **to** $\mathrm{I}\ \mathrm{I}')\ \langle C' : Cl \,|\, Inh : \mathrm{I}' \rangle$ **if** $(C \neq C')$

This search corresponds to breadth-first, left-first traversal of the inheritance graph.

6.5 Guarded Statements

Guards represent potential processor release points. Guards may be boolean or reply guards. When a guard is encountered, the execution continues if the guard is enabled:

$\langle o : Ob \,|\, Pr : \mathit{await}\ g; \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A}, EvQ : \mathrm{Q} \rangle$
$\longrightarrow$
$\langle o : Ob \,|\, Pr : \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A}, EvQ : \mathrm{Q} \rangle$ **if** $\mathit{enabled}(g, (\mathrm{A}, \mathrm{L}), \mathrm{Q})$

Enabledness is defined by induction over the construction of guards by the predicate

$$\mathit{enabled}(n?, \mathrm{D}, \mathrm{Q}) = n\ \mathit{in}\ \mathrm{Q} \qquad \mathit{enabled}(b, \mathrm{D}, \mathrm{Q}) = \mathit{eval}(b, \mathrm{D})$$

where D denotes a state, and the function *in* checks whether a completion message corresponding to the given label value is in the message queue Q. Enabledness is extended to statement lists, considering the head statement, and considering unguarded statements as enabled. When a non-enabled guard is encountered, the active process is *suspended* on the process queue:

$\langle o : Ob \,|\, Pr : \mathrm{S}, PrQ : \mathrm{W}, Lvar : \mathrm{L}, Att : \mathrm{A}, EvQ : \mathrm{Q} \rangle$
$\longrightarrow$
$\langle o : Ob \,|\, Pr : \varepsilon, PrQ : (\mathrm{W}\ \langle \mathrm{S}, \mathrm{L} \rangle), Lvar : \varepsilon, Att : \mathrm{A}, EvQ : \mathrm{Q} \rangle$ **if** $\mathit{not\ enabled}(\mathrm{S}, (\mathrm{A}; \mathrm{L}), \mathrm{Q})$

where $\langle \mathrm{S}, \mathrm{L} \rangle$ denotes the process with statements S and local state L. If there is no active process, a suspended process can be *reactivated* if it is enabled:

$\langle o : Ob \,|\, Pr : \varepsilon, PrQ : \langle \mathrm{S}, \mathrm{L} \rangle\ \mathrm{W}, Lvar : \mathrm{L}', Att : \mathrm{A}, EvQ : \mathrm{Q} \rangle$
$\longrightarrow$
$\langle o : Ob \,|\, Pr : \mathrm{S}, PrQ : \mathrm{W}, Lvar : \mathrm{L}, Att : \mathrm{A}, EvQ : \mathrm{Q} \rangle$ **if** $\mathit{enabled}(\mathrm{S}, (\mathrm{A}, \mathrm{L}), \mathrm{Q})$

This rule allows any enabled process to continue because PrQ is a multiset.

6.6 Object Creation and Attribute Instantiation

Object creation results in a new object with a unique identifier. The new object makes an initial blocking call to its *run* method (if present in the class), thereby initiating active object behavior and leaving the programmer in control of defining the initial release point. New object identifiers are created by concatenating tokens n from the unbounded set *Tok* to the class name. The identifier is returned to the object which initiated the object creation.

$\langle o : Ob \mid Pr : v := new\ C(In); \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A} \rangle\ \langle C : Cl \mid Att : \mathrm{A}', Tok : n \rangle$
$\longrightarrow$
$\langle o : Ob \mid Pr : v := newid; \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A} \rangle\ \langle C : Cl \mid Att : \mathrm{A}', Tok : next(n) \rangle$
$\langle newid : Ob \mid Cl : C, Pr : run, PrQ : \varepsilon, Lvar : \varepsilon, Att : \varepsilon, Lab : 1, EvQ : \varepsilon \rangle$
$inherit(newid, \varepsilon)$ **to** $C(eval(In, (\mathrm{A}, \mathrm{L})))$

Here, *newid* denotes the new identifier. Before the new object can be activated, its initial state must be created. This is done by collecting attribute lists, which consist of program variables bound to initial expressions, from the classes inherited by C. The initial expressions must be reduced to values and bound to the program variables in the state. Class parameters and inherited attributes provide a mechanism to pass values to the initial expressions of the inheritance list in a class. The variables bound by the class parameters are stored first in the attribute list of a class in the textual order.

An *inherit* message, which sends an object identifier and a substitution to a class inheritance list, causes the inheritance tree to be traversed in a right-first depth-first order, while dynamically accumulating all inherited attributes and their initializing expressions, passing on appropriate class parameters as stated in the inheritance lists. The traversal results in a list of attributes with initializing expressions, which are evaluated by *evalS* from left to right and delivered to the new object. The attribute list is ordered such that the attributes of a superclass precede those of a subclass, for all classes above the class of the object. Consequently, the type system can guarantee that all variables occurring in an initial expression of a program variable v have been instantiated before v is instantiated.

$inherit(o, \mathrm{IA})$ **to** $nil = inherited(evalS((self \mapsto o)\ \mathrm{IA}), \varepsilon)$ **to** o

$inherit(o, \mathrm{IA})$ **to** $(\mathrm{I}\ C(In))\ \ \langle C : Cl \mid Inh : \mathrm{I}', Att : \mathrm{IA}' \rangle$
$\quad = inherit(o, (pass(\mathrm{IA}', In)\ \mathrm{IA}))$ **to** $(\mathrm{I}\ \mathrm{I}')\ \ \langle C : Cl \mid Inh : \mathrm{I}', Att : \mathrm{IA}' \rangle$

The auxiliary function *pass* passes class parameters, given as expressions, to an attribute list and $evalS(\mathrm{IA}, \mathrm{A})$ evaluates attributes in IA from left to right, given a state A.

$pass(\mathrm{IA}, \varepsilon) = \mathrm{IA}$
$pass(((v \mapsto e)\ \mathrm{IA}), e'\ \mathrm{E}')(v \mapsto e')\ pass(\mathrm{IA}, \mathrm{E}')$

$evalS(\varepsilon, \mathrm{A}) = \varepsilon$
$evalS((v \mapsto e)\ \mathrm{IA}, \mathrm{A}) = (v \mapsto eval(e, \mathrm{A}))\ evalS(\mathrm{IA}, (v \mapsto eval(e, \mathrm{A}))\ \mathrm{A})$

The resulting state is consumed by the new object by the equation

$(inherited(\mathrm{A})$ **to** $o)\ \ \langle o : Ob \mid Att : \varepsilon \rangle = \langle o : Ob \mid Att : \mathrm{A} \rangle$

Notice again that the use of equations enables a new object to be created and initialized in a single rewriting step.

In the presence of multiple inheritance, a class C may inherit a superclass several times. The equation

$\mathrm{A}\ (v \mapsto e)\ \mathrm{A}'\ (v \mapsto e') = \mathrm{A}\ (v \mapsto e)\ \mathrm{A}'$

on attribute lists ensures that an attribute is only stored once. Thus multi-inheritance of the same class is the same as inheriting the class once, keeping the leftmost instantiation. Duplicate classes may be achieved by class renaming in inheritance lists.

7 Related Work

Formal models clarify the intricacies of object orientation and may thus contribute to better programming languages in the future, making programs easier to understand, maintain, and reason about. Work on object calculi such as the ς-calculus [1] capture object-oriented features such as self-reference, encapsulation, and method calls. Concurrent object calculi [18,13] extend these mechanisms to multithreaded and distributed systems, but the complexities of class inheritance are not addresses in [1,18,13]. A concurrent object calculus with single inheritance is presented by Laneve [26]. Methods of superclasses are accessible and virtual binding is addressed due to a careful renaming discipline. A denotational semantics for single inheritance with similar features is studied by Cook and Palsberg [9]. Multiple inheritance is not addressed in these works.

Formalizations of multiple inheritance in the literature are usually based on the *objects-as-records* paradigm. This approach focuses on subtyping issues related to subclassing, but issues related to method binding are not easily captured. Even access to methods of superclasses is not addressed in Cardelli's denotational semantics of multiple inheritance [6]. Rossi, Friedman, and Wand [34] propose a formal definition of multiple inheritance based on *subobjects*, a run-time data structure used for virtual pointer tables [25,38]. This formalism focuses on compile time issues and does not clarify multiple inheritance at the abstraction level of the programming language.

Multiple inheritance is supported in languages such as C++ [38], CLOS [12], Eiffel [31], POOL [2], and Self [7]. As discussed in Sect. 2.1, horizontal name conflicts in C++, POOL, and Eiffel are removed by explicit resolution, after which the inheritance graph may be linearized. Multiple dispatch, or multi-methods[12], gives a more powerful binding mechanism, but does not handle the problems considered here, since they appear even for methods without any parameters. Also reasoning about multi-methods is difficult in case of redefinition.

A natural semantics for virtual binding in Eiffel is proposed in [3]. This work is similar in spirit to ours and models the binding mechanism at the abstraction level of the program, capturing Eiffel's renaming mechanism. Mixin-based inheritance [4] and traits [35] also depend upon linearization to be merged correctly into the single inheritance chain. Linearization changes the parent-child relationship between classes in the inheritance hierarchy [36]. Consequently understanding, e.g., method binding quickly becomes difficult.

Full Maude [8] and the Join-calculus [15] model multiple inheritance by disjoint union of methods. Name ambiguity lets method definitions compete for selection. The definition selected when an ambiguously named method is called, may be nondeterministically chosen. Alternatively, programmer control may be improved if inherited classes are ordered [7,12], resulting in a deterministic binding strategy. However, the

ordering of superclasses may result in surprising but "correct" behavior. The example of Sect. 5 displays such surprising behavior regardless of how the inherited classes are ordered.

The dynamically typed prototype-based language Self [7] proposes an elegant *prioritized binding strategy* to solve this problem, although a formal semantics is not given. The strategy is based on combining ordered and unordered multiple inheritance. Each superclass is annotated with a priority, and many superclasses may have the same priority. A name is only ambiguous if it occurs in two superclasses with the same priority, in which case a class related to the caller class is preferred. However, explicit class priorities may have surprising effects in large class hierarchies: names may become ambiguous through inheritance. If neither class is related to the caller the binding does not succeed, resulting in a method-not-understood error.

The *pruned binding strategy* proposed in this paper solves these issues without the need for manually declaring (equal) class priorities and without the possibility of method-not-understood errors: Calls are only bound to intended method redefinitions. The new binding strategy seems particularly useful during system maintenance to avoid introducing unintentional errors in evolving class hierarchies, as targeted by the Creol language [23]. In particular, we have given an operational semantics based on dynamic and distributed traversal of the available classes, rather than through virtual pointer tables. Our approach may therefore be combined with dynamic constructs for changing the class inheritance structure, such as adding a class C and enriching an existing class with C as a new superclass, which could be useful in open reconfigurable systems.

8 Conclusion

The treatment of ambiguous naming in object-oriented languages with multiple inheritance is unsettled. Disallowing naming ambiguities when inheriting multiple superclasses imposes undesirable restrictions with regard to, e.g., programming flexibility and code maintenance. Ordering inherited classes solves ambiguities by fixing the binding strategy above a given class. However, virtual binding combined with a fixed order may lead to surprising but "correct" effects. This paper has proposed the *pruned binding strategy* to ensure that overriding is intended. This strategy dynamically restricts the ordered inheritance graph depending on the context of the call, using the concept of constrained method call ($m < C$). This construct is also useful for fine grained programmer control of virtual binding in the case of multiple inheritance. The pruned binding strategy and constrained method calls remove unintended effects of ordered inheritance while ensuring that binding will always succeed. The binding strategy is combined with intentional redirection through qualified references and with redefinition in the subclass. In this paper, an operational semantics for the proposed binding strategy has been given in rewriting logic. Although the formalization is given in the setting of Creol, the mechanisms presented here could easily be lifted to another setting.

Acknowledgment. The authors would like to thank Stein Krogdahl for interesting discussions on multiple inheritance and virtual binding. The comments of the FMCO anonymous referees have improved the presentation.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, New York, NY, 1996.
2. P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In N. Meyrowitz, editor, *Proc. of the Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 161–168. ACM Press, Oct. 1990.
3. I. Attali, D. Caromel, and S. O. Ehmety. A natural semantics for Eiffel dynamic binding. *ACM Transactions on Programming Languages and Systems*, 18(6):711–729, 1996.
4. G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proc. of the Conf. on Object-Oriented Programming: Systems, Languages, and Applications / Eur. Conf. on Object-Oriented Programming*, pages 303–311. ACM Press 1990.
5. K. B. Bruce, A. Schuett, R. van Gent, and A. Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems*, 25(2):225–290, 2003.
6. L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2-3):138–164, 1988.
7. C. Chambers, D. Ungar, B.-W. Chang, and U. Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *Lisp and Symb. Computation*, 4(3):207–222, 1991.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.
9. W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, Nov. 1994.
10. W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *17th Symp. on Principles of Programming Languages (POPL'90)*, pages 125–135. ACM Press, Jan. 1990.
11. O.-J. Dahl and K. Nygaard. Class and subclass declarations. In J. Buxton, editor, *Simulation Programming Languages*, pages 158–174. North-Holland, 1968. Reprinted in M. Broy and E. Denert, eds., *Software Pioneers — Contributions to Software Engineering*, Springer, 2002.
12. L. G. DeMichiel and R. P. Gabriel. The common lisp object system: An overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Eur. Conf. on Object-Oriented Programming (ECOOP'87)*, LNCS 276, pages 151–170. Springer, 1987.
13. P. Di Blasio and K. Fischer. A calculus for concurrent objects. In U. Montanari and V. Sassone, editors, *7th Intl. Conf. on Concurrency Theory (CONCUR'96)*, LNCS 1119, pages 655–670. Springer, Aug. 1996.
14. C. Fischer and H. Wehrheim. Behavioural subtyping relations for object-oriented formalisms. In T. Rus, editor, *8th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST 2000)*, LNCS 1816, pages 469–483. Springer, 2000.
15. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the join calculus. *Journal of Logic and Algebraic Programming*, 57(1-2):23–69, 2003.
16. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.
17. J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer, 2000.
18. A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In U. Nestmann and B. C. Pierce, editors, *High-Level Concurrent Languages (HLCL'98)*, volume 16(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.
19. E. B. Johnsen and O. Owe. A compositional formalism for object viewpoints. In B. Jacobs and A. Rensink, editors, *Proc. 5th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)*, pages 45–60. Kluwer, Mar. 2002.

20. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. In *Proc. 2nd Intl. Conf. on Software Engineering and Formal Methods (SEFM'04)*, pages 188–197. IEEE Press, Sept. 2004.
21. E. B. Johnsen and O. Owe. Object-oriented specification and open distributed systems. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, LNCS 2635, pages 137–164. Springer, 2004.
22. E. B. Johnsen and O. Owe. Inheritance in the presence of asynchronous method calls. In *Proc. 38th Hawaii Intl. Conf. on System Sciences (HICSS'05)*. IEEE Press, Jan. 2005.
23. E. B. Johnsen, O. Owe, and I. Simplot-Ryl. A dynamic class construct for asynchronous concurrent objects. In *Proc. 7th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, LNCS 3535, pages 15–30. Springer, June 2005.
24. J. L. Knudsen. Name collision in multiple classification hierarchies. In S. Gjessing and K. Nygaard, editors, *Eur. Conf. on Object-Oriented Programming (ECOOP'88)*, LNCS 322, pages 93–109. Springer, 1988.
25. S. Krogdahl. Multiple inheritance in Simula-like languages. *BIT*, 25(2):318–326, 1985.
26. C. Laneve. Inheritance in concurrent objects. In H. Bowman and J. Derrick, editors, *Formal methods for distributed processing – a survey of object-oriented approaches*, pages 326–353. Cambridge University Press, 2001.
27. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.
28. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. The MIT Press, Cambridge, Mass., 1993.
29. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
30. J. Meseguer and G. Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In D. A. Basin and M. Rusinowitch, editors, *Proc. of the 2nd Intl. Joint Conf. on Automated Reasoning (IJCAR 2004)*, LNCS 3097, pages 1–44. Springer, 2004.
31. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, NJ., 1997.
32. G. Milicia and V. Sassone. The inheritance anomaly: ten years after. In *Proc. of the Symp. on Applied Computing*, pages 1267–1274. ACM Press, 2004.
33. E. Najm and J.-B. Stefani. A formal semantics for the ODP computational model. *Computer Networks and ISDN Systems*, 27:1305–1329, 1995.
34. J. G. Rossie Jr., D. P. Friedman, and M. Wand. Modeling subobject-based inheritance. In P. Cointe, editor, *10th Eur. Conf. on Object-Oriented Programming (ECOOP'96)*, LNCS 1098, pages 248–274. Springer, July 1996.
35. N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In L. Cardelli, editor, *Proc. 17th Eur. Conf. on Object-Oriented Programming (ECOOP 2003)*, LNCS 2743, pages 248–274. Springer, 2003.
36. A. Snyder. Inheritance and the development of encapsulated software systems. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 165–188. The MIT Press, 1987.
37. N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proc. Fifth Intl. Conf. on Software Reuse (ICSR5)*, pages 206–215. IEEE Press, 1998.
38. B. Stroustrup. Multiple inheritance for C++. *Computing Systems*, 2(4):367–395, Dec. 1989.
39. E. Tempero and R. Biddle. Simulating multiple inheritance in Java. *The Journal of Systems and Software*, 55(1):87–100, Nov. 2000.

Observability, Connectivity, and Replay in a Sequential Calculus of Classes*

Erika Ábrahám[2], Marcello M. Bonsangue[3], Frank S. de Boer[4], Andreas Grüner[1], and Martin Steffen[1]

[1] Christian-Albrechts-University Kiel, Germany
[2] Albert-Ludwigs-University Freiburg, Germany
[3] University Leiden, The Netherlands
[4] CWI Amsterdam, The Netherlands

Abstract. Object calculi have been investigated as semantical foundation for object-oriented languages. Often, they are object-based, whereas the mainstream of object-oriented languages is *class-based.*

Considering classes as part of a component makes instantiation a possible interaction between component and environment. As a consequence, one needs to take *connectivity* information into account.

We formulate an operational semantics that incorporates the connectivity information into the scoping mechanism of the calculus. Furthermore, we formalize a notion of equivalence on traces which captures the uncertainty of observation cause by the fact that the observer may fall into separate groups of objects. We use a corresponding trace semantics for full abstraction wrt. a simple notion of observability. This requires to capture the notion of *determinism* for traces where classes may be instantiated into more than one instance during a run and showing thus twice an equivalent behavior (doing a "replay"), a problem absent in an object-based setting.

Keywords: class-based object-oriented languages, formal semantics, determinism, full abstraction.

1 Introduction

Classes are a structuring concept for object-oriented languages such as *Java* or $C^{\#}$. This raises the question what the semantics of a program is when considering classes as composition units. A simple, elegant, and common semantical approach is to take an observational point of view: two program fragments are equal, if, when put in any possible context, no difference can be seen. Starting from a simple notion of observation, [10] presented a fully abstract trace semantics for a multithreaded *object* calculus, i.e., a language without classes; [2] generalized the result by taking *classes* into account.

* Part of this work has been financially supported by the IST project Omega (IST-2001-33522) and the NWO/DFG project Mobi-J (RO 1122/9-1/2).

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 296–316, 2005.
© Springer-Verlag Berlin Heidelberg 2005

In this paper, we re-address the problem in a *single-threaded* setting. This is interesting for two reasons. First, simplifying the language does not simplify the problem *per se*. Certain complications in connection with concurrency certainly get simpler, e.g., by absence of race conditions. On the other hand, new complications arise. In particular, with one thread only, the language becomes *deterministic* which needs to be accounted for in the description of the semantics. Secondly, concentrating on a single thread allows to understand the semantical impact of classes more clearly and independently from the orthogonal aspects of concurrency.

One key observation is that in the presence of classes one needs to take connectivity information into account, i.e., the way objects may have knowledge of each other, to characterize the observable behavior. In particular, unconnected environment objects can neither determine the absolute order of interaction, nor can they exchange information to compare object identities. Furthermore, with a deterministic language, one needs to capture the notion of *determinism* for traces where classes may be instantiated into more than one instance during a run and showing thus twice an equivalent behavior (doing a "replay"), a problem absent in an object-based setting.

Overview. The paper is organized as follows. We start in Section 2 with an informal account of the semantics and the underlying intuitions. Section 3 contains the syntax of the calculus and a sketch of its semantics. In particular, the notions of lazy instantiation and connectivity of objects are formalized. Afterwards, Section 4 elaborates on the trace semantics and in particular an equivalence relation on traces capturing the uncertainty of observation in a class-based setting. In Section 5 we fix the notion of observation and state the full abstraction result. Section 6 concludes with related and future work.

2 Observability and Classes

This section presents on an intuitive level the consequences of incorporating *classes* into the observational set-up.

2.1 Cross-Border Instantiation and Connectivity

The observational set-up separates *classes* into component and environment classes. Hence, not only calls and returns are exchanged at the interface between component and environment, but instantiation requests, as well.

If, for instance, the component creates an instance of an environment class, the interaction between the component and the newly created object can entail observable effects in the future, as the code of the object is externally provided and therefore this interaction belongs to the externally visible observer-program behavior. Hence, instances of environment classes belong to the environment, and dually those of internal classes to the component. However, in the above situation, the *reference* to the new external object is kept at the creator for

the time being. So if the component instantiates two objects o_2 and o_3 of the environment, the situation looks informally as in Figure 1, where the dotted bubbles indicate the scope of o_2, respectively of o_3.

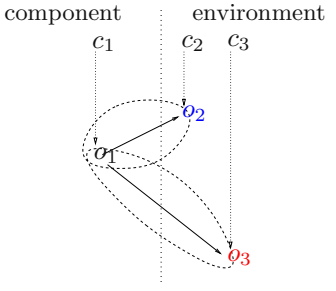


Fig. 1. Instances of external classes

For an exact account of the semantics, the inability of o_2 and o_3 to be in contact must be accounted for. More generally, the semantics must contain a representation of which object can possibly be in contact with others, i.e., an overapproximation of the heap's *connectivity.* Sets of objects which can possibly be in contact with each other form therefore equivalence classes of names —we call them *cliques*— and the semantics must include a representation of them. New cliques can be created, as new objects can be instantiated without contact to others, and furthermore cliques can merge, if the component leaks the identity of a member of one clique to a member of another.

2.2 Different Observers and Order of Events

That the observer may fall into separate cliques of unconnected objects has implications for what can be observed. First of all, the absolute *order of events* cannot be determined, as the observer cliques may not be able to coordinate. For instance, the environment or observer, split into 1 and 2 on the right-hand sides of the three scenarios of Figure 2 cannot distinguish between the three variants of the component on the respective left-hand sides. Note that the clique structure is dynamic, since communication can merge previously separate observer cliques. After merging, the now joint clique, indicated by the big "bubble" containing 1 and 2 in Figure 3, can coordinate and thus observe the order of further interaction, but the order of past interaction cannot be reconstructed. I.e., in Figure 3, the three components, i.e., the components on the left-hand side of Figure 3(a) – 3(c) respectively, are observably equivalent.

2.3 Classes as Generators of Objects, Replay, and Determinism

Classes are generators for objects, and two instances of a class are *"identical up-to their identity"* i.e., they have the same behavior up-to renaming. If the trace of a component contains a certain behavior of an object (or more generally of a clique of objects), then it is unavoidable that the component shows a trace where the equivalent behavior is realized by a second instance of the object (or object clique): each behavior can be "replayed" on a fresh instance. With the possibility of cross-border instantiation, the component can create more than one equivalent instances of its observer, which perform equivalently.

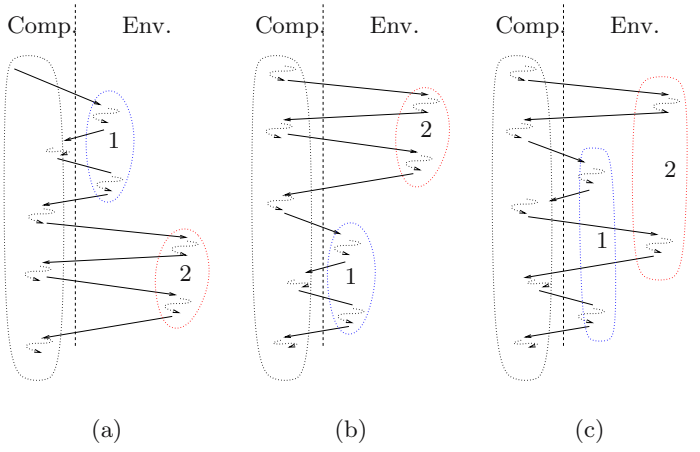


Fig. 2. Order of interaction

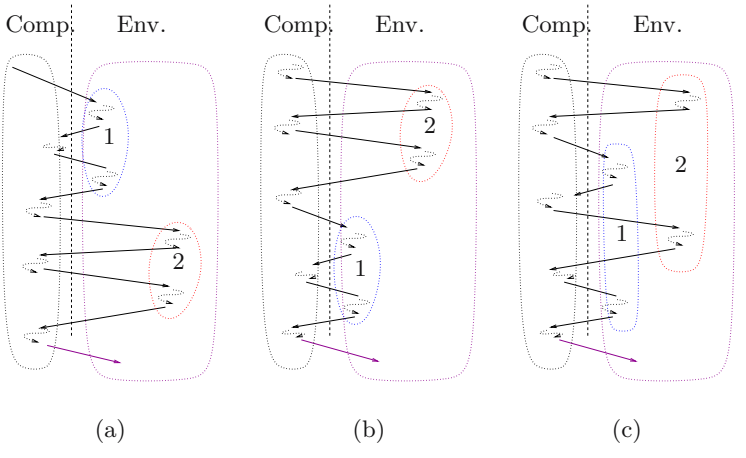
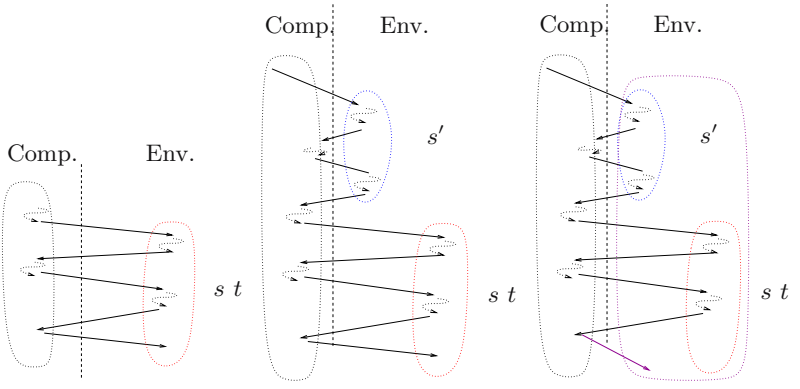


Fig. 3. Order of interaction and merging

Consider Figures 4(a) and 4(b). The second one resembles Figure 3(a) before the merge. This time, however, we assume, that the interaction s' with the first clique is a prefix of the longer $s\ t$ up-to renaming. If $s\ t$ is a possible behavior of the system, then clearly also scenario 4(b). One can use the argument also in the reverse direction: if 4(b) is possible, then so is 4(a); in other words, both behaviors are equivalent.

If afterwards the observers are merged (cf. Figure 4(c)), this scenario clearly differs from the one where the interaction s' with the formerly separate clique is missing. Unlike in the situation of Figure 3, where the order of the previously separate cliques could not be enforced in retrospect, the merging here allows to compare the different identities (but of course still not the order).



(a) Scenario $s\ t$ (b) Replay (c) Merged

Fig. 4. Replay and merging

The possibility to create more than one instance from a class has a further impact when dealing with deterministic programs in the single-threaded setting. If a class is instantiated twice, its instances must behave "the same" up-to renaming, i.e., when confronted with the same input, show the same reaction. For instance, the shorter trace s' of Figure 4(b) is not only possible, given $s\ t$, but the clique on the left of 4(b) can do *nothing else* than what does the one on the right, when stimulated by the same input from the component. The scenario used environment cliques for illustration, but the same arguments apply to component cliques, as well.

3 A Single-Threaded Calculus with Classes

Concentrating on the semantical issues, we only sketch the syntax and ignore typing issues as they are rather standard and similar to [2]. Indeed, the calculus is a restriction to single threaded programs of the one used in [2], which in turn is an extension of the concurrent object calculus from [6,10] namely by adding classes.

A program is given by a collection of classes where a class $c[\![O]\!]$ carries a name c and defines the implementation of its methods and fields.[1] An object $o[c, F]$ stores the current value of the fields or instance variables and keeps a reference to the class it instantiates. A method $\varsigma(n{:}c).\lambda(x_1{:}T_1, \ldots, x_k{:}T_k).t$ provides the method body abstracted over the ς-bound "self" parameter and the formal parameters of the method [1]. Besides named objects and classes, the dynamic

[1] For names, we will generally use o and its syntactic variants as names for objects, c for classes, and n when being unspecific, for instance in Table 1.

Table 1. Abstract syntax

$C ::= \mathbf{0} \mid C \parallel C \mid \nu(n{:}T).C \mid n[\![O]\!] \mid n[n, F] \mid \natural\langle t\rangle$	program
$O ::= F, M$	object
$M ::= l = m, \ldots, l = m$	method suite
$F ::= l = f, \ldots, l = f$	fields
$m ::= \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t$	method
$f ::= \varsigma(n{:}T).\lambda().v \mid \varsigma(n{:}T).\lambda().\mathit{stop}$	field
$t ::= v \mid \mathit{stop} \mid \mathit{let}\, x{:}T = e\, \mathit{in}\, t$	thread
$e ::= t \mid \mathsf{if}\ v = v\ \mathsf{then}\ e\ \mathsf{else}\ e$	expression
$\quad \mid \ v.l(v, \ldots, v) \mid v.l := f \mid \mathit{new}\, n$	
$v ::= x \mid n$	value

configuration of a program contains one single thread $\natural\langle t\rangle$ as active entity.[2] A thread basically is either a value or a sequence of expressions, notably method calls (written $v.l(\vec{v})$) and the creation of new objects $\mathit{new}\, c$ where c is a class name. Furthermore we will use f specifically for instance variables or fields, we use $f = v$ for field variable declaration, field access is written as $x.f$, and field update as $x.f := v$.

The *operational semantics* is given in two levels: *internal* steps whose effect is confined within a component, and those with external effect. The *external* behavior of a component is given in terms of labeled transitions describing the communication at the interface of an *open* program. For the completeness of the semantics, it is crucial ultimately to consider only communication traces realizable by an actual program context which, together with the component, yields a well-typed closed program.

Being concerned with the dynamic connectivity among objects, we omit in this paper most of the typing aspects, e.g., that transmitted values need to adhere to the static typing assumptions, that only publicly known objects can be called from the outside, and the like, since this part is rather standard and also quite similar to the one in [10].

3.1 Operational Semantics

We start with component internal steps in the following section; Section 3.1.2 contains the small-step semantics describing the component-environment interaction.

3.1.1 Internal Steps. The internal steps are given in Table 2, where we distinguish between confluent steps, written $\rightsquigarrow$, and other internal transitions, written $\xrightarrow{\tau}$.[3]

[2] The $\natural$-symbol is only meant to distinguish the syntactic entity t from a running thread $\natural\langle t\rangle$.

[3] In the single-threaded setting, the distinction is not too important, since at any time at most one reduction step is enabled. It may nevertheless enhance understanding to conceptually distinguish between side-effect free steps and those that may lead to race conditions when executed in the presence of other threads.

Table 2. Internal steps

$$\natural\langle let\, x{:}T = v\, in\, t\rangle \rightsquigarrow \natural\langle t[v/x]\rangle \qquad \textsc{Red}$$

$$\natural\langle let\, x_2{:}T_2 = (let\, x_1{:}T_1 = e_1\, in\, e)\, in\, t\rangle \rightsquigarrow \natural\langle let\, x_1{:}T_1 = e_1\, in\, (let\, x_2{:}T_2 = e\, in\, t)\rangle \qquad \textsc{Let}$$

$$\natural\langle let\, x{:}T = (\mathsf{if}\ v = v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\, in\, t\rangle \rightsquigarrow \natural\langle let\, x{:}T = e_1\, in\, t\rangle \qquad \textsc{Cond}_1$$

$$\natural\langle let\, x{:}T = (\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\, in\, t\rangle \rightsquigarrow \natural\langle let\, x{:}T = e_2\, in\, t\rangle \qquad \textsc{Cond}_2$$

$$\natural\langle let\, x{:}T = stop\, in\, t\rangle \rightsquigarrow \natural\langle stop\rangle \qquad \textsc{Stop}$$

$$c[\![F, M]\!] \parallel \natural\langle let\, x{:}c = new\, c\, in\, t\rangle \rightsquigarrow$$
$$c[\![F, M]\!] \parallel \nu(o{:}c).(o[c, F] \parallel \natural\langle let\, x{:}c = o\, in\, t\rangle) \qquad \textsc{NewO}_i$$

$$c[\![F, M]\!] \parallel o[c, F'] \parallel \natural\langle let\, x{:}T = o.l(\vec{v})\, in\, t\rangle \xrightarrow{\tau}$$
$$c[\![F, M]\!] \parallel o[c, F'] \parallel \natural\langle let\, x{:}T = M.l(o)(\vec{v})\, in\, t\rangle \qquad \textsc{Call}_i$$

$$o[c, F] \parallel \natural\langle let\, x{:}T = o.f := v\, in\, t\rangle \xrightarrow{\tau} o[c, F.f := v] \parallel \natural\langle let\, x{:}T = o\, in\, t\rangle \qquad \textsc{FUpdate}$$

Table 3. Structural congruence

$$\mathbf{0} \parallel C \equiv C$$
$$C_1 \parallel C_2 \equiv C_2 \parallel C_1 \qquad (C_1 \parallel C_2) \parallel C_3 \equiv C_1 \parallel (C_2 \parallel C_3)$$
$$C_1 \parallel \nu(n{:}T).C_2 \equiv \nu(n{:}T).(C_1 \parallel C_2)$$
$$\nu(n_1{:}T_1).\nu(n_2{:}T_2).C \equiv \nu(n_2{:}T_2).\nu(n_1{:}T_1).C$$

The reduction relations from above are used modulo *structural congruence,* which captures the algebraic properties of parallel composition and the hiding operator. The basic axioms for $\equiv$ are shown in Table 3 where in the fourth axiom, n does not occur free in C_1.

3.1.2 External Steps. While the component-internal steps are fairly standard and straightforward, the external semantics is more complex. With the goal of full-abstraction in mind it is necessary to ultimately characterize the interaction between component and environment which is realizable by *some* program (cf. the legal traces from Section 5.2). To ease the full abstraction argument, we formulate the semantics under the assumption that the component together with the (absent or abstracted) environment gives a well-formed program adhering to the syntactical and the context-sensitive restrictions of the language at hand.

So the interface behavior is phrased in an assumption-commitment framework and based on three orthogonal abstractions:

- a static abstraction, i.e., the type system (largely omitted here);
- an abstraction of the stacks of recursive method invocations, representing the recursive and reentrant nature of method calls in a multi-threaded setting;
- finally an abstraction of the *heap topology,* approximating potential connectivity of objects and threads. The heap topology is dynamic in that new objects may be created and tree structured in that previously separate groups of objects may merge.

Connectivity Contexts and Cliques. As discussed, in the presence of internal and external classes and cross-border instantiation, the semantics must contain a representation of the object connectivity. The external semantics is formalized as labeled transitions between judgments of the form

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta \ , \tag{1}$$

where $\Delta; E_\Delta$ are the *assumptions* about the environment of the component C and $\Theta; E_\Theta$ the *commitments*. The assumptions consist of a part Δ concerning the existence (plus static typing information) of *named entities* in the environment. For the book-keeping of which objects of the environment have been told which identities, a well-typed component must take into account the *relation* of object names from the assumption context Δ amongst each other, and the knowledge of objects from Δ about those exported by the component, i.e., those from Θ. In analogy to the name contexts Δ and Θ, E_Δ expresses assumptions about the environment, and E_Θ commitments of the component:

$$E_\Delta \subseteq \Delta \times (\Delta + \Theta) \ . \tag{2}$$

and dually $E_\Theta \subseteq \Theta \times (\Theta + \Delta)$, where $\times$ denotes the pairing and $+$ the disjoint combination of Δ and Θ. We write $o_1 \hookrightarrow o_2$ ("o_1 may know o_2") for pairs from these relations. Without full information about the complete system, the component must make worst-case assumptions concerning the proliferation of knowledge, which are represented as the *reflexive, transitive,* and *symmetric* closure of the $\hookrightarrow$-pairs of objects *from* Δ. Given Δ, Θ, and E_Δ, we write $\leftrightharpoons$ for this closure, i.e.,

$$\leftrightharpoons \triangleq \ (\hookrightarrow\!\downarrow_\Delta \cup \hookleftarrow\!\downarrow_\Delta)^* \subseteq \Delta \times \Delta \ . \tag{3}$$

In the definition, $\downarrow_\Delta$ stands for the projection of the relation onto the restricted domain Δ. Note that we close $\hookrightarrow$ only wrt. environment objects, but not wrt. objects at the *interface,* i.e., the part of $\hookrightarrow \subseteq \ \Delta \times \Theta$. We also need the union $\leftrightharpoons \cup \leftrightharpoons; \hookrightarrow \subseteq \Delta \times (\Delta + \Theta)$, where the semicolon denotes relational composition. We write $\leftrightharpoons\!\!\hookrightarrow$ for that union. As judgment, we use $\Delta; E_\Delta \vdash o_1 \leftrightharpoons o_2 : \Theta$, resp. $\Delta; E_\Delta \vdash o_1 \leftrightharpoons\!\!\hookrightarrow o_2 : \Theta$. For Θ, E_Θ, and Δ, the definitions are applied dually.

The relation $\leftrightharpoons$ is an equivalence relation on the objects from Δ and partitions them into equivalence classes. We call a set of object names from Δ (or dually from Θ) such that for all objects o_1 and o_2 from that set, $\Delta; E_\Delta \vdash o_1 \leftrightharpoons o_2 : \Theta$, a *clique*, and if we speak of *the* clique of an object we mean the equivalence class.

External Steps. The external semantics is given by transitions between $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ judgments in Table 5. Besides internal steps a component exchanges information with the environment via *calls* and *returns* (cf. Table 4).

Table 4. Labels

$\gamma ::= \langle call\ o.l(\vec{v})\rangle \mid \langle return(v)\rangle \mid \nu(n{:}T).\gamma$	basic labels
$a ::= \gamma? \mid \gamma!$	receive and send labels

To formulate the external communication, we need to augment the syntax by two additional expressions o_1 *blocks for* o_2 and o_2 *returns to* $o_1\, v$. The first one denotes a method body in o_1 waiting for a return from o_2, and dually the second expression returns v from o_2 to o_1. Furthermore, we augment the syntax of the method definitions accordingly, such that each method call is preceded by an annotation of the caller; i.e., instead of $\varsigma(self{:}c).\lambda(\vec{x}{:}\vec{T}).(\ldots x.l(\vec{y})\ldots)$ we write $\varsigma(self{:}c).\lambda(\vec{x}{:}\vec{T}).(\ldots self\, x.l(\vec{y})\ldots)$.

Connectivity Assumptions and Commitments. As for the relationship of communicated values, incoming and outgoing communication play dual roles: E_Θ over-approximates the actual connectivity of the component, while the assumption context E_Δ is consulted to exclude impossible combinations of incoming values. *Incoming* calls update the commitment context E_Θ in that it remembers that the callee o_2 now knows (or rather may know) the arguments $\vec{v}$. For incoming communication (cf. rules CallI and RetI) we require that the sender is acquainted with the transmitted arguments.

For the role of the caller identity o_1: The antecedent of the call-rules requires, that the caller o_1 is acquainted with the callee o_2 and with all of the arguments. However, the caller is *not* transmitted in the label which means that it remains anonymous to the callee.[4] To gauge, whether an incoming call is possible and to adjust the book-keeping about the connectivity appropriately. With the sole exception of the initial (external) step, the scope of at least *one* object of the calling clique must have escaped to the component, for otherwise there would be now way of the caller to address o_2 as callee. In other words, for at least one object o_1 from the clique of the actual caller (which remains anonymous), the judgment $\Delta \vdash o_1 : c$ holds prior to the call, where the judgment —the typing rules are not shown here— asserts that object o_1 carries type c according to the *environment* (and not the component) context.

While E_Δ imposes restrictions for incoming communication, the commitment context E_Θ is *updated* when receiving new information. For instance in CallI, the commitment $\acute{E}_\Theta$ after reception marks that now the callee o_2 is acquainted with the received arguments. For *outgoing* communication, the E_Δ and E_Θ play

[4] Of course, the caller may transmit its identity to the callee as argument, but this does not reveal to the callee who "actually" called. Indeed, the actual identity of the caller is not needed; it suffices to know the *clique* of the caller. As representative for the clique, an equivalence class of object identities, we simply pick one object.

Table 5. External steps

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ \langle call\ o_2.l(\vec{v})\rangle?\quad dom(\Delta', \Theta') \subseteq fn(\langle call\ o_2.l(\vec{v})\rangle)\quad \vdash \Delta, \Theta : static \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; o_2 \hookrightarrow \vec{v}\quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; \odot \hookrightarrow (\Delta', \Theta')\quad \Delta \vdash \odot \\ ; \acute{\Theta} \vdash o_2 : c_2\quad ; \Delta; \Theta \vdash c_2 : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!]\quad ; \acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T}\quad \acute{\Delta}; \acute{E}_\Delta \vdash \odot \leftrightharpoons \vec{v}, o_2 : \acute{\Theta} \end{array}}{\begin{array}{l} \Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash C \parallel C(\Theta') \parallel \natural\langle let\ x{:}T = o_2.l(\vec{v})\ in\ o_2\ returns\ to \odot\ x\rangle : \acute{\Theta}; \acute{E}_\Theta \end{array}}\ \textsc{CallI}_0$$

$$\frac{\begin{array}{c} a = \nu(\Theta', \Delta').\ \langle call\ o_2.l(\vec{v})\rangle!\quad (\Theta', \Delta') = fn(\langle call\ o_2.l(\vec{v})\rangle \cap \Phi\quad \acute{\Phi} = \Phi \setminus (\Theta', \Delta') \\ \acute{\Delta} \vdash o_2 : c_2\quad \vdash \Delta, \Theta : static \\ \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_2 \hookrightarrow \vec{v}\qquad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; E(C, \Theta') \end{array}}{\begin{array}{l} \Delta; E_\Delta \vdash \nu(\Phi).(C \parallel \natural\langle let\ x{:}T = \odot\ o_2.l(\vec{v})\ in\ t\rangle) : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel \natural\langle let\ x{:}T = \odot\ blocks\ for\ o_2\ in\ t\rangle) : \acute{\Theta}; \acute{E}_\Theta \end{array}}\ \textsc{CallO}_0$$

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ \langle call\ o_2.l(\vec{v})\rangle?\quad dom(\Delta', \Theta') \subseteq fn(\langle call\ o_2.l(\vec{v})\rangle) \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; o_2 \hookrightarrow \vec{v}\quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_1 \hookrightarrow (\Delta', \Theta') \\ ; \acute{\Theta} \vdash o_2 : c_2\quad ; \Delta; \Theta \vdash c_2 : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!]\quad ; \acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T} \\ \acute{\Delta}; \acute{E}_\Delta \vdash o_1 \leftrightharpoons \vec{v}, o_2 : \acute{\Theta}\quad t_{blocked} = let\ x'{:}T' = o_2'\ blocks\ for\ o_1\ in\ t \end{array}}{\begin{array}{l} \Delta; E_\Delta \vdash C \parallel \natural\langle t_{blocked}\rangle : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash C \parallel C(\Theta') \parallel \natural\langle let\ x{:}T = o_2.l(\vec{v})\ in\ o_2\ returns\ to\ o_1\ x; t_{blocked}\rangle : \acute{\Theta}; \acute{E}_\Theta \end{array}}\ \textsc{CallI}$$

$$\frac{\begin{array}{c} a = \nu(\Theta', \Delta').\ \langle return(v)\rangle!\qquad (\Theta', \Delta') = fn(v) \cap \Phi\qquad \acute{\Phi} = \Phi \setminus (\Theta', \Delta') \\ \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_1 \hookrightarrow v\qquad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; E(C, \Theta') \end{array}}{\begin{array}{l} \Delta; E_\Delta \vdash \nu(\Phi).(C \parallel \natural\langle let\ x{:}T = o_2\ returns\ to\ o_1\ v\ in\ t\rangle) : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel \natural\langle t\rangle) : \acute{\Theta}; \acute{E}_\Theta \end{array}}\ \textsc{RetO}$$

$$\frac{\begin{array}{c} a = \nu(\Theta', \Delta').\ \langle call\ o_2.l(\vec{v})\rangle!\quad (\Theta', \Delta') = fn(\langle call\ o_2.l(\vec{v})\rangle \cap \Phi\quad \acute{\Phi} = \Phi \setminus (\Theta', \Delta') \\ \acute{\Delta} \vdash o_2 : c_2\quad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_2 \hookrightarrow \vec{v}\qquad \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; E(C, \Theta') \end{array}}{\begin{array}{l} \Delta; E_\Delta \vdash \nu(\Phi).(C \parallel \natural\langle let\ x{:}T = o_1\ o_2.l(\vec{v})\ in\ t\rangle) : \Theta; E_\Theta \xrightarrow{a} \\ \acute{\Delta}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel \natural\langle let\ x{:}T = o_1\ blocks\ for\ o_2\ in\ t\rangle) : \acute{\Theta}; \acute{E}_\Theta \end{array}}\ \textsc{CallO}$$

$$\frac{\begin{array}{c} a = \nu(\Delta', \Theta').\ \langle return(v)\rangle?\qquad dom(\Delta', \Theta') \subseteq fn(v) \\ \acute{\Theta}; \acute{E}_\Theta = \Theta; E_\Theta + \Theta'; o_1 \hookrightarrow v\qquad \acute{\Delta}; \acute{E}_\Delta = \Delta; E_\Delta + \Delta'; o_2 \hookrightarrow (\Delta', \Theta') \\ ; \Delta \vdash o_2 : c_2\quad ; \Delta; \Theta \vdash c_2 : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!]\quad ; \acute{\Delta}, \acute{\Theta} \vdash v : T\quad \acute{\Delta}; \acute{E}_\Delta \vdash o_2 \leftrightharpoons v : \acute{\Theta} \end{array}}{\begin{array}{l} \Delta; E_\Delta \vdash C \parallel \natural\langle let\ x{:}T = o_1\ blocks\ for\ o_2\ in\ t\rangle : \Theta; E_\Theta \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash C \parallel \natural\langle t[v/x]\rangle : \acute{\Theta}; \acute{E}_\Theta \end{array}}\ \textsc{RetI}$$

$$\frac{\Delta \vdash c : T}{\Delta; E_\Delta \vdash \natural\langle let\ x{:}c = new\ c\ in\ t\rangle : \Theta; E_\Theta \rightsquigarrow \Delta; E_\Delta \vdash \nu(o_3{:}c).\natural\langle let\ x{:}c = o_3\ in\ t\rangle : \Theta; E_\Theta}\ \textsc{NewO}_{lazy}$$

dual roles. In the respective rules, $E(\acute{C}, \Theta')$ stands for the actual connectivity of the component after the step, which needs to be made public in the commitment context, in case new names escape to the environment.

In case of the very first interaction, either an incoming or outgoing call (cf. rules $\textsc{CallI}_0$ or $\textsc{CallO}_0$), we take $\odot$ as the source of the call, which is assumed to be resident either in the environment or the component. Furthermore, at the beginning, no objects are visible yet across the border which is asserted by $\mathit{static}(\Delta, \Theta)$. The remaining premises of the form $;\Delta \vdash n : T$ or similar deal with static typing issue, i.e., guaranteeing subject reduction. We omit the formalization of the static typing system here, as it is straightforward.

Scoping and Lazy Instantiation. In the explanation so far, we omitted the handling of bound names, in particular bound object references. In the presence of classes, a possible interaction between component and environment is instantiation. Without constructor methods and assuming an infinite heap space, instantiation itself has no immediate, observable side-effect. An observable effect is seen only at the point when the object is accessed.

Rule $\textsc{NewO}_{lazy}$ describes the local instantiation of an external class. Instead of exporting the newly created name of the object plus the object itself immediately to the environment, the name is kept local until, if ever, it gets into contact with the environment. When this happens, the new instance will not only become known to the environment, but the object will also be instantiated in the environment.

For incoming calls, for instance, the binding part is of the form (Δ', Θ') where we mean by convention, that Δ' are the object name being added to Δ, and analogously for Θ' and Θ. The distinction is based on the class types which are never transmitted. For the object names in the incoming communication, Δ' contains the external references which are freshly introduced to the component by scope extrusion. Θ' on the other hand are the objects which are *lazily instantiated* as side-effect of this step, and which are from then on part of the component. In the rules, the newly instantiated objects are denoted as $C(\Theta')$.

Note that whereas the acquaintance of the caller with the arguments transmitted free is checked against the current assumption, acquaintance with the ones transmitted bound is added to the assumption context.

4 Trace Semantics and Ordering on Traces

Next we present the semantics for well-typed components, which takes the *traces* of the program fragment as starting point. A trace t is a sequence of external steps, i.e., given by $\Xi_1 \vdash C_1 \stackrel{s}{\Longrightarrow} \Xi_2 \vdash C_2$.

The clique structure of the environment influences what is observable and the fact that the observer falls into a number of independent groups of objects increases the "uncertainty of observation". In Section 2, we informally discussed two reasons responsible for this effect. One is that the clique of objects can only observe the order of events *projected* to its own members but not the relative order among separate cliques. Secondly, separate observers cannot cooperate to *compare identities.*

For the definition, we need to connect the labels of a trace to the clique they belong to. With the exception of the callee of a call, the communication labels

do not carry information about the identity of the communication partners (cf. Table 4). Given a trace of past interaction, which adheres to a strict call-return discipline and which is strictly alternating between input and output, it contains enough information to determine the communication partners.

4.1 Balance Conditions

We start with auxiliary definitions concerning the parenthetic nature of calls and returns of a legal trace (cf. Definition 1). The definition is similar to the one from [10]. It is easy to see that, starting from an initial configuration, the operational semantics from Section 3.1.2 assures strict alternation of incoming and outgoing communication and additionally that there is no return without a preceding matching call. Later, we will need this property of traces for the characterization of legal traces.

Definition 1 (Balance). *The balance of a sequence s is given by the rules of Table 6, where the dual rules for balanced$^-$ are omitted. We write $\vdash s : balanced$ if $\vdash s : balanced^+$ or $\vdash s : balanced^-$. We call a (not necessarily proper) prefix*

Table 6. Balance

$$\frac{}{\vdash \epsilon : balanced^+}\ \text{B-EMPTY}^+$$

$$\frac{\vdash s_1 : balanced^+ \qquad \vdash s_2 : balanced^+ \qquad s_1, s_2 \neq \epsilon}{\vdash s_1\ s_2 : balanced^+}\ \text{B-II}$$

$$\frac{\vdash s : balanced^-}{\vdash \nu(\Phi).\langle call\ o_r.l(\vec{v})\rangle!\ s\ \nu(\Phi').\langle return(v)\rangle? : balanced^+}\ \text{B-OI}$$

of a balanced trace weakly balanced. *We write $\vdash s : wbalanced^+$ if the trace is weakly balanced and if the last label is an incoming communication or if s is empty; dually for $\vdash s : wbalanced^-$.*

The function pop on traces is defined as follows:

1. *pop $s = \bot$, if s is balanced.*
2. *pop $(s_1 a s_2) = s_1 a$ if $a = \nu(\Delta, \Theta).\ \langle call\ o_2.l(\vec{v})\rangle?$ and s_2 is balanced$^+$.*
3. *pop $(s_1 a s_2) = s_1 a$ if $a = \nu(\Delta, \Theta).\ \langle call\ o_2.l(\vec{v})\rangle!$ and s_2 is balanced$^-$.*

Note that the definition of *pop*, when defined, yields a unique value. Especially, the three cases are mutually exclusive and in case 2. resp. 3, the requirement that s_2 is balanced determines $s_1 a$ uniquely.

Based on a balanced past, the following definition formalizes the notion of source and target of a communication event at the end of a trace with the help of the function *pop*.

Definition 2 (Sender and receiver). *Let $r\ a$ be a balanced trace. Sender and receiver of a after history r are defined by mutual recursion and pattern matching over the following cases:*

$$\begin{aligned} sender(\nu(\Phi).\langle call\ o_2.l(\vec{v})\rangle!) &= \odot \\ sender(r'\ a'\ \nu(\Phi).\langle call\ o_2.l(\vec{v})\rangle!) &= receiver(r'\ a') \\ sender(r'\ a'\ \nu(\Phi).\langle return(l(\vec{v}))\rangle!) &= receiver(pop(r'\ a')) \end{aligned}$$

$$\begin{aligned} receiver(r\ \nu(\Phi).\langle call\ o_2.l(\vec{v})\rangle!) &= o_2 \\ receiver(r\ \nu(\Phi).\langle return(\vec{v})\rangle!) &= sender(pop(r)) \end{aligned}$$

For $a = \nu(\Phi)\langle call\ o_2.l(\vec{v})\rangle?$ resp. $a = \nu(\Phi).\langle return(\vec{v})\rangle?$, the definition is dual.

4.2 Equivalences

Now given a global trace, its projection onto one particular clique of objects as given at the end of the trace is defined straightforwardly by induction on the length of the trace. We write $[o]_{/E_\Delta}$ for the equivalence class of objects according to E_Δ, i.e., the clique in connection with o, or in general just shorter $[o]$ when E_Δ is clear from the context.

Definition 3 (Projection). *Assume as trace $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{s}{\Longrightarrow} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$ and let $\acute{\Delta}$ contain at least one object reference, then the projection of s onto a clique $[o]$ of environment objects according to $\acute{\Delta}; \acute{E}_\Delta$ is written as $s \downarrow_{[o]}$ and defined by induction on the length of s: $s \downarrow_{[o]}$ is defined as the first component of $(s, \Phi) \downarrow_{[o]}$, where $\Phi = \Delta, \Theta$, and the projection of $(s, \Phi) \downarrow_{[o]}$ is given by Table 7. The definition of the projection onto a* component *clique is defined dually.*

The projection of the empty trace surely is empty (rule P-Empty). For output actions in P-Out$_1$ and P-Out$_2$ we distinguish according to the receiver, i.e., the callee in case of a call resp. the caller in case of a return. If the receiver is not involved in the communication, the label is "projected out"; dually for incoming communication. More interesting is P-Out$_2$: fresh names are not only the globally fresh ones Φ'_1, but also the locally fresh ones Φ'_2. The situation for incoming new names is *not symmetric!* It is simpler as we need not distinguish between locally and globally new names: Everything that the clique has created in isolation is globally new as well as locally new.

Besides "local freshness" we have to cater for the fact that the *order* of events cannot be determined by separate observers, i.e., we need to formalize the ideas illustrated in Section 2.2. We do this by a notion of swappability, where sub-sequences can be reordered when indistinguishable by the environment. This means the definition takes into account the worst-case estimations from E_Δ about the clique structure of the environment, which we indicate by the

Table 7. Projection to an environment clique

$$\frac{}{(\epsilon, \Phi) \downarrow_{[o]} = (\epsilon, \Phi)}\ \text{P-Empty}$$

$$\frac{(t, \Phi) \downarrow_{[o]} = (t', \Phi') \qquad \mathit{receiver}(t\gamma!) \notin [o]}{(t\gamma!, \Phi) \downarrow_{[o]} = (t', \Phi')}\ \text{P-Out}_1$$

$$\frac{\begin{array}{c}\Phi_2' = \mathit{fn}(\nu(\Phi_1').\gamma) \setminus \Phi' \\ (t, \Phi) \downarrow_{[o]} = (t', \Phi') \qquad \mathit{receiver}(t\gamma!) \in [o]\end{array}}{(t\ \nu(\Phi_1').\gamma!, \Phi) \downarrow_{[o]} = (t'\ \nu(\Phi_1', \Phi_2').\gamma!, (\Phi', \Phi_1', \Phi_2'))}\ \text{P-Out}_2$$

$$\frac{(t, \Phi) \downarrow_{[o]} = (t', \Phi') \qquad \mathit{sender}(t\gamma?) \notin [o]}{(t\gamma?, \Phi) \downarrow_{[o]} = (t', \Phi')}\ \text{P-In}_1$$

$$\frac{(t, \Phi) \downarrow_{[o]} = (t', \Phi') \qquad \mathit{sender}(t\gamma?) \in [o]}{(t\ \nu(\Phi'').\gamma?, \Phi) \downarrow_{[o]} = (t'\ \nu(\Phi'').\gamma?, (\Phi', \Phi''))}\ \text{P-In}_2$$

subscript[5] Δ. The dual version of the relation, written $\asymp_\Theta$, takes into account the clique structure of the component. It captures the possible reorderings of a given behavior of the component.

Whether or not the order of two actions in a trace is indistinguishable depends on clique situation of the environment at the point where the actions occur. Therefore we generalize the judgment $\Delta; E_\Delta \vdash\ o_1 \leftrightharpoons o_2 : \Theta$ from Section 3 to express acquaintance *after* executing some trace.

Definition 4 (Dynamic acquaintance). *Assume* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$. *We write* $\Delta; E_\Delta \vdash s \rhd o_1 \leftrightharpoons o_2 : \Theta; E_\Theta$, *if* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{s}{\Longrightarrow} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta$ *and* $\acute{\Delta}; \acute{E}_\Delta \vdash o_1 \leftrightharpoons o_2 : \acute{\Theta}$. *The notation is used analogously for* $\leftrightharpoons\!\!\hookrightarrow$.

We use the definition analogously for subsequences of a trace, i.e., given $\Delta; E_\Delta \vdash C : \Theta; E_\Theta \stackrel{st_1t_2u}{\Longrightarrow} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \Theta; E_\Theta$, we write $\Delta; E_\Delta \vdash s \rhd t_1 \leftrightharpoons t_2 : \Theta$ if there exists a communication partner[6] o_1 of the environment mentioned in t_1 and a communication partner o_2 from t_2 acquainted according to Definition 4.

Definition 5 (Swapping). *The relation* $\asymp_\Delta$ *on traces is given as the reflexive, symmetric, and transitive closure of the rules of Table 8. The two rules silently assume that the traces involved are weakly balanced. The relation* $\asymp_\Theta$ *is defined dually.*

[5] The Δ is meant just as indication, that swappability is interpreted from the perspective of the environment, not as a concrete argument of the definition of $\asymp$.

[6] Sender or receiver depending on whether the action is incoming or outgoing.

Table 8. Swapping

$$\frac{\Xi \vdash s \rhd t_1 \not\leftrightharpoons t_2 \qquad \vdash t_1 : \mathit{balanced}}{\Xi \vdash s\ \nu(\Phi).t_1 t_2 u \asymp_\Delta s\nu(\Phi).t_2 t_1 u}\ \text{E-SwapB}_\Delta$$

$$\frac{\Xi \vdash s \rhd t_1 \not\leftrightharpoons t_2 \qquad \vdash t_1, t_2 : \mathit{wbalanced}}{\Xi \vdash s\ \nu(\Phi).t_1 t_2 \asymp_\Delta s\nu(\Phi).t_2 t_1}\ \text{E-SwapW}_\Delta$$

The definition of $\asymp_\Delta$ distinguishes between swapping of two neighboring subsequences in the middle of a trace (rule E-SwapB$_\Delta$) and at the end (rule E-SwapW$_\Delta$). In case of E-SwapB$_\Delta$, we require that one of the subsequences, in the rule t_1, is in itself balanced, i.e., without the preceding and trailing "contexts" s resp. $t_2 u$. Note that t_2 is not required to be balanced, as well (but the rule can be applied symmetrically); the swapping, however, must preserve overall weak balance. That balance requirement for t_1 is needed illustrates the following consideration: Take for instance the right-hand side st_2t_1u, then moving t_2 after an unbalanced (for instance only weakly balanced) t_1 may (re-)connect returns in t_2 to unanswered calls in t_1. Similarly, returns in u may be reconnected, which means that they belong to a different environment cliques when comparing st_1t_2u and st_2t_1u. This may lead to observably different behavior. Requiring that one of the sub-sequences is balanced avoids this effect. Similar considerations imply that for swapping sub-sequences at the end, we must require *weak* balance (cf. rule E-SwapW$_\Delta$). Note that it is not sufficient that only *one* of the sub-sequences involved is weakly balanced.

Remains the formalization of the fact that different instances of the same class, or more generally different cliques identical up-to their identities, do not count as adding new behavior to the system, i.e., next we formalize the intuition from Section 2.3. The equivalence relation $\asymp$ from above is extended to consider two behaviors as equivalent if one clique is just a "replay" (up-to renaming of behavior) already witnessed in the trace. In other words: a trace can be equivalently extended by an additional action, if the behavior of the extended clique is contained as behavior of another clique already, i.e., in the form of a prefix, for which we write $\preccurlyeq$. Note that the prefix is understood up-to α-renaming.

Definition 6 (Swapping and replay). *The relation $\asymp_\Delta$ on traces is given by the reflexive, transitive, and symmetric closure of the relation given in Table 9. The relation $\asymp_\Theta$ is defined dually.*

We can now define the order on traces as follows.

Definition 7. *$\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \sqsubseteq_{trace} \Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$, if the following holds. If $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \overset{s}{\Longrightarrow}$, then $\Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta \overset{t}{\Longrightarrow}$ such that $\Delta; E_\Delta \vdash t : det_\Delta \Theta; E_\Theta$.*

Table 9. Swapping and replay

$$\frac{\begin{array}{c} receiver(s\gamma!) = o \\ s\gamma! \downarrow_{[o]} \preccurlyeq\ s \downarrow_{[o']} \end{array}}{\Delta; E_\Delta \vdash s\gamma! \asymp_\Delta s : trace\,\Theta; E_\Theta}\ \text{E-REO}_\Delta \qquad \frac{\begin{array}{c} sender(s\gamma?) = o \\ s\gamma? \downarrow_{[o]} \preccurlyeq\ s \downarrow_{[o']} \end{array}}{\Delta; E_\Delta \vdash s\gamma? \asymp_\Delta s : trace\,\Theta; E_\Theta}\ \text{E-REI}_\Delta$$

$$\frac{\Delta; E_\Delta \vdash s \asymp_\Delta t : \Theta; E_\Theta}{\Delta; E_\Delta \vdash s \asymp_\Delta t : \Theta; E_\Theta}\ \text{E-SWAP}_\Delta$$

5 Full Abstraction

After fixing the notion of observation, we address one core problem for establishing the connection between the trace preorder and the contextual preorder, namely the characterization of legal traces, i.e., the traces which are realizable by a component together with an *arbitrary* (but well-formed, well-typed ...) context. Especially in the single-threaded setting this requires to capture deterministic traces.

5.1 Notion of Observation

Full abstraction is a comparison between two semantics, where the reference semantics to start from is traditionally *contextually* defined and based on a some notion of *observability.*

As starting point we choose, as [10], a (standard) notion of semantic equivalence or rather semantic implication —one program allows at least the observations of the other— based on a particular, simple form of contextual observation: being put into a context, the component, together with the context, reaches a defined point, which is counted as the successful observation. Being deterministic, there is no need to distinguish whether the program "may" reach the point of observation or "must" reach it. A context $\mathcal{C}[_]$ is a program "with a hole". In our setting, the hole is filled with a program fragment consisting of a *component* C in the syntactical sense, i.e., consisting of the parallel composition of (named) classes, named objects, and named threads, and the context is the rest of the programs such that $\mathcal{C}[C]$ gives a well-typed *closed* program $\Delta; E_\Delta \vdash C' : \Theta; E_\Theta$, where closed means that it can be typed in the empty contexts, i.e., $\vdash C' : ()$.

To report success, we assume an external class with a particular success reporting method. So assume a class c_b of type $[\![succ : () \to none]\!]$, abbreviated as barb. A component C *strongly barbs on* c_b, written $C \downarrow_{c_b}$, if $C \equiv \nu(\vec{n}{:}\vec{T}, b{:}c_b).C' \parallel \natural\langle let\ x{:}none = b.succ()\ in\ t\rangle$, i.e., the call to the success-method of an instance of c_b is enabled. Furthermore, C *barbs on* c_b, written $C \Downarrow_{c_b}$, if it can reach a point which strongly barbs on c_b, i.e., $C \Longrightarrow C' \downarrow_{c_b}$. We can now define observable preorder [7] similar as in [10]. Since the programs are deterministic, the distinction between a "may" and a "must" success disappears.

Definition 8 (Observable preorder). *Assume* $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta$ *and* $\Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$. *Then* $\Delta; E_\Delta \vdash C_1 \sqsubseteq_{obs} C_2 : \Theta; E_\Theta$, *if* $(C_1 \parallel C) \Downarrow_{c_b}$ *implies* $(C_2 \parallel C) \Downarrow_{c_b}$ *for all* $\Theta, c_b{:}\mathsf{barb}; E_\Theta \vdash C : \Delta; E_\Delta$.

5.2 Legal Traces

As mentioned, we must characterize which traces, the "legal" ones, can occur at all, and again the crucial difference to the object-based case is to take connectivity into account to exclude impossible combinations of transmitted object names and threads. Furthermore, we need to filter out non-deterministic ones in the single-threaded setting.

The legal traces are specified by a system for judgments of the form $\Delta; E_\Delta \vdash r \rhd s : \mathit{trace}\,\Theta; E_\Theta$ stipulating that under the type and relational assumptions Δ and E_Δ and with the commitments Θ and E_Θ, the trace s is legal. The rules are shown in Table 10. The premises of the form $;\acute{\Theta} \vdash o_2 : c_2$, $;\Delta, \Theta \vdash c_2 : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!]$, and $;\acute{\Delta}, \acute{\Theta} \vdash \vec{v} : \vec{T}$, e.g., as mentioned in rule L-CALLI, check that message exchange respects the static typing assumptions.

Table 10. Legal traces

$$\Delta; E_\Delta \vdash r \rhd \epsilon : \mathit{trace}\,\Theta; E_\Theta \qquad \text{L-EMPTY}$$

$$\frac{\vdash r \rhd o_s \xrightarrow{a} o_r \quad \Xi = \Xi + o_s \xrightarrow{a} o_r \quad \acute{\Delta}, \acute{\Theta} \vdash \lfloor a \rfloor : ok \quad \Xi \vdash o_s \overset{\lfloor a \rfloor}{\hookrightarrow} o_r : ok \quad \Delta, \Theta \not\vdash \mathit{static} \quad a = \nu(\Phi').\,\langle \mathit{call}\ o_r.l(\vec{v})\rangle? \quad \acute{\Delta}; \acute{E}_\Delta \vdash r\ a \rhd s : \mathit{trace}\,\acute{\Theta}; \acute{E}_\Theta}{\Delta; E_\Delta \vdash r \rhd a\ s : \mathit{trace}\,\Theta; E_\Theta} \text{ L-CALLI}$$

$$\frac{\vdash r \rhd o_s \xrightarrow{a} o_r \quad \Xi = \Xi + o_s \xrightarrow{a} o_r \quad \Xi \vdash o_s \overset{\lfloor a \rfloor}{\hookrightarrow} o_r : ok \quad \Delta', \Theta' \vdash \lfloor a \rfloor : ok \quad a = \nu(\Phi').\,\langle \mathit{return}(v)\rangle? \quad \acute{\Delta}; \acute{E}_\Delta \vdash r\ a \rhd s : \mathit{trace}\,\acute{\Theta}; \acute{E}_\Theta}{\Delta; E_\Delta \vdash r \rhd a\ s : \mathit{trace}\,\Theta; E_\Theta} \text{ L-RETI}$$

$$\frac{\vdash \epsilon \rhd \odot \xrightarrow{a} o_r \quad \Xi = \Xi + \odot \xrightarrow{a} o_r \quad \acute{\Delta}, \acute{\Theta} \vdash \lfloor a \rfloor : ok \quad \Xi \vdash \odot \overset{\lfloor a \rfloor}{\hookrightarrow} o_r : ok \quad \Delta_0, \Theta_0 \vdash \mathit{static} \quad \Delta \vdash \odot \quad a = \nu(\Delta', \Theta').\,\langle \mathit{call}\ o_r.l(\vec{v})\rangle? \quad \acute{\Delta}; \acute{E}_\Delta \vdash a \rhd s : \acute{\Theta}; \acute{E}_\Theta}{\Delta_0 \vdash \epsilon \rhd a\ s : \mathit{trace}\,\Theta_0} \text{ L-CALLI}_0$$

The premise $\Delta \vdash r \rhd a : \Theta$ asserts that after r, the action a is enabled.

Definition 9 (Enabledness). *Given a method call* $\gamma = \nu(\Phi).\langle \mathit{call}\ o_2.l(\vec{v})\rangle$. *Then call-*enabledness *of* γ *after the history* r *and in the contexts* Δ *and* Θ *is defined as:*

$$\Delta; E_\Delta \vdash r \rhd \gamma? : \Theta; E_\Theta \ \textit{if}\ \mathit{pop}\ r = \bot \ \textit{and}\ \Delta \vdash \odot \ \textit{or}\ \mathit{pop}\ r = r'\gamma'! \tag{4}$$

$$\Delta; E_\Delta \vdash r \rhd \gamma! : \Theta; E_\Theta \ \textit{if}\ \mathit{pop}\ r = \bot \ \textit{and}\ \Delta \vdash \odot \ \textit{or}\ \mathit{pop}\ r = r'\gamma'? \tag{5}$$

For return labels $\gamma = \nu(\Phi).\langle return(v)\rangle$, $\Delta; E_\Delta \vdash r \rhd \gamma! : \Theta; E_\Theta$ *abbreviates* $pop\, r = r'\nu(\Phi').\langle call\ o_2.l(\vec{v})\rangle?$, *and dually for incoming returns* $\gamma?$.

We also say, the thread is *input-call enabled* after r if $\Delta \vdash r \rhd \gamma? : \Theta$ for some incoming call label, respectively *input-return* enabled in case of an incoming return label. The definitions are used dually for output-call enabledness and output-return enabledness. When leaving the kind of communication unspecified we just speak of input-enabledness or output-enabledness. Note that return-enabledness implies call-enabledness, but not vice versa.

Being single-threaded, the language is *deterministic,* i.e., given a configuration, the next operational step is determined (up-to possible renamings). This is not only a fact about the global system behavior, but also —and more interestingly in our context— tells us that two instances of the same class, when stimulated by the same input history must react identically, up-to renaming (cf. also the discussion in Section 2.3). We thus need a characterization of deterministic traces to define when a trace is legal or not.

The issue has various aspects. That we can speak of a single trace being deterministic or not is a consequence of having classes with the possibility of cross-border instantiation and thus the possible presence of separate cliques of objects. Only then, different behaviors of "the same" object or more generally "the same" clique can show up in the trace, where the non-deterministic ones need to be filtered out to obtain an adequate characterization of the legal traces. Furthermore, the intuition "determinism means the same reaction to the same stimulus" needs some fleshing out. The past of a clique is the projection of the global trace onto the clique, which is, as usual, considered only up-to α-renaming. Furthermore, the dynamic nature of the clique structure has to be taken into account; for instance, the histories corresponding to Figure 3(a) – 3(c) are to be considered equivalent because the order of events of previously separate sub-cliques of a given clique cannot be reconstructed in retrospect.

The mentioned ideas are captured in the $\asymp$ relation, which we can use in the following definition.

Definition 10 (Deterministic trace). *Given the label* $a = \gamma!$ *and a trace* ra *with* $\Delta \vdash r \rhd a : \Theta$. *The trace* r *can be extended deterministically by* a, *written* $\Delta \vdash r \rhd a : det_\Theta\ \Theta$, *if the following holds:*

$$\begin{array}{l} \Delta; E_\Delta \vdash ra \asymp_\Theta r : \Theta; E_\Theta \quad or \\ \text{there does not exist a label } b \text{ with } \quad \Delta; E_\Delta \vdash rb \asymp_\Theta r : \Theta; E_\Theta \end{array} \tag{6}$$

The definition for incoming communications a *is dual, and especially refers to* $\asymp_\Delta$ *instead of* $\asymp_\Theta$.

Note that the condition from Equation (6) does not in itself guarantee determinism for the trace; if the shorter r is deterministic, it preserves determinism when extending the trace, which is the way, the check is used in the legal trace system. We use the judgment $\Delta; E_\Delta \vdash r \rhd a : det_\Theta\ \Theta; E_\Theta$ to combine enabledness and the output determinism requirement for the next action in a single assertion. Dually we use det_Δ for input determinism for incoming communication. We write

also $\Delta; E_\Delta \vdash s : det_\Delta \Theta; E_\Theta$ resp. $\Delta; E_\Delta \vdash t : det_\Theta \Theta; E_\Theta$, when the whole trace is deterministic wrt. the environment, resp. wrt. component.

5.3 Soundness and Completeness

The proof that the observational order coincides with the order on traces given in Definition 7 has two directions: compared to $\sqsubseteq_{obs}$, the relation $\sqsubseteq_{trace}$ is neither too abstract (soundness) nor too concrete (completeness). For lack of space, we simply state the soundness result here.

For correspondence of the two notions is guaranteed only when assuming that the environment behaves deterministic. Therefore we refine the definition of $\sqsubseteq_{trace}$ from Definition 7, in that we explicitly require that the traces compared by $\sqsubseteq_{trace}$ are deterministic wrt. the environment; we write $\sqsubseteq^{det}_{trace}$ for that relation. The reason is that the external operational semantics of Table 5 results in deterministic behavior as far as the component is concerned —one cannot program non-deterministic behavior with the given syntax— but not for the environment. One could have checked deterministic environment behavior in the assumptions of the operational rule; the price for this more exact representation of possible behavior would have been to augment the semantics to contain the *history* of past interaction concerning the environment behavior, in a similar way as we have done when formalizing the legal traces.

Proposition 1 (Soundness). *If* $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \sqsubseteq^{det}_{trace} \Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$*, then* $\Delta; E_\Delta \vdash C_1 \sqsubseteq_{obs} C_2 : \Theta; E_\Theta$*.*

Completeness asserts the reverse direction:

Proposition 2 (Completeness). *If* $\Delta; E_\Delta \models C_1 \sqsubseteq_{obs} C_2 : \Theta; E_\Theta$*, then* $\Delta; E_\Delta \vdash C_1 : \Theta; E_\Theta \sqsubseteq^{det}_{trace} \Delta; E_\Delta \vdash C_2 : \Theta; E_\Theta$*.*

At the heart, completeness is a constructive argument: given a trace s, construct a component C_s that exhibits the trace s and moreover realize it *exactly*. Restricted to deterministic traces, the proof is rather similar to the one for the multi-threaded case and rests on the ability to compose a component and an environment, performing complementary traces, into one global program (plus the dual property of decomposition). Indeed, the very same construction could be used in the single-threaded setting as in the multi-threaded setting. However, the absence of concurrency allows to simplify the construction, in particular, one can leave out the code that assures mutual exclusion, when accessing objects, resp. cliques of objects.

6 Conclusion

Related Work. Smith [17] presents a fully abstract model for *Object-Z*, an object-oriented extension of the *Z* specification language. called the *complete-readiness* model, related to the readiness model of Olderog and Hoare. [18] investigates full abstraction in an object calculus with subtyping. The setting is

a bit different from the one as used here as the paper does not compare a contextual semantics with a denotational one, but a semantics by translation with a direct one. The paper considers neither concurrency nor aliasing. Recently, Jeffrey and Rathke [11] extended their work on trace-based semantics from an object-based setting to a core of *Java*, called *JavaJr*, including classes and subtyping. However, their semantics avoids the issue of object connectivity by using a notion of *package*. Cf. also [14]. [5] tackles the problem of full abstraction and observable component behavior and connectivity in a UML-setting. Unlike this contribution, [5] features concurrency

Future Work. The trace semantics together with the equivalence relation capturing the undefinednes of order of interacting with separe cliques is a "*tree*" semantics. As illustrated also by the informal examples of Section 2, the semantics more precisely can be understood as a forest of interactions, where each tree represents one current clique of objects. As shown in this paper, the cliques can be dynamically created and the branching structure is cause by merging of cliques. We are currently working on a *direct* tree representation of the semantics. The resulting semantic is is simpler as it can do without the secondary notion of equivalence relation on traces, and furthermore one can avoid an explicit representation of object connectivity as. However, e.g., the derivation system for legal traces gets more involved in that it must reflect the branching structure.

Game theory has in recent years been successfully employed for (fully abstract) semantics of open system ("game semantics"). Cf. for instance [3] for an introduction. It seems interesting to capture our set-up especially the connectivity contexts in a game semantical framework.

Acknowledgements. We thank Harald Fecher and Marcel Kyas for stimulating discussions on various aspects of this topic.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.
2. E. Ábrahám, M. M. Bonsangue, F. S. de Boer, and M. Steffen. Object connectivity and full abstraction for a concurrent calculus of classes. In Li [12], pages 38–52.
3. S. Abramsky. Algorithmic game semantics: A tutorial introduction. In Schichtenberg and Steinbruggen [16], pages 21–47.
4. M. Bosangue, F. S. de Boer, W.-P. de Roever, and S. Graf, editors. *Proceedings of the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, Lecture Notes in Computer Science. Springer-Verlag, 2005. To appear.
5. F. S. de Boer, M. Bonsangue, M. Steffen, and E. Ábrahám. A fully abstract trace semantics for UML components. In Bosangue et al. [4]. To appear.
6. A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In Nestmann and Pierce [13].
7. M. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

8. IEEE. *Thirteenth Annual Symposium on Logic in Computer Science (LICS) (Indiana)*. Computer Society Press, July 1998.
9. IEEE. *Seventeenth Annual Symposium on Logic in Computer Science (LICS) (Copenhagen, Denmark)*. Computer Society Press, July 2002.
10. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In LICS'02 [9].
11. A. Jeffrey and J. Rathke. Java Jr.: A fully abstract trace semantics for a core Java language. In Sagiv [15], pages 423–438.
12. Z. Li, editor. *Proceedings of the First International Colloquium on Theoretical Aspects of Computing, ICTAC 2004*, volume 3407 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
13. U. Nestmann and B. C. Pierce, editors. *HLCL '98: High-Level Concurrent Languages (Nice, France, September 12, 1998)*, volume 16.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.
14. J. Rathke. A fully abstract trace semantics for a core Java language (preliminary title). In Bosangue et al. [4]. To appear.
15. M. Sagiv, editor. *Proceedings of ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
16. H. Schichtenberg and R. Steinbruggen, editors. *Proof and System Reliability, Summer School (Marktoberdorf, Germany, 2001)*, Series F: Computer and System Sciences. NATO Advanced Study Institute, Kluwer Academic Publishers, 2001.
17. G. P. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, Department of Computer Science, University of Queensland, Oct. 1992.
18. R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In LICS'98 [8].

Timing Analysis and Timing Predictability Extended Abstract

Reinhard Wilhelm*

Informatik, Universität des Saarlandes, Saarbrücken

Abstract. Hard real-time systems need methods to determine upper bounds for their execution times, usually called worst-case execution times. This paper explains the principles of our Timing-Analysis methods, which use Abstract Interpretation to predict the system's behavior on the underlying processor's components and use Integer Linear Programming to determine a worst-case path through the program. Under the assumption that non-trivial systems are subject of the analyses, exhaustive analyses can not be performed and some uncertainty about the system's behavior remains. Uncertainty, i.e., lack of information about a system's execution states incurs cost in terms of precision of the upper and lower bounds on the execution times. Some cost figures are given for missing information of different types. These are measured in machine clock cycles. It is (intuitively) argued, that component-based software design and the use of middleware may induce intolerable costs in terms of precision.

1 Execution-Time Variability

Hard real-time systems need methods to determine upper bounds for their execution times, usually called worst-case execution times, WCET. Based on these bounds, a schedulability analysis must check whether the underlying hardware is fast enough to execute the system's task such that they all finish before their deadlines. This problem is nontrivial because performance-enhancing architectural features such as caches, pipelines, and all kinds of speculation destroy the traditional compositional methods to determine bounds on execution time. These used so-called Timing Schemata [Sha89] for the statements of the programming language describing how to use the bounds for the constituents of the statements to compose bounds for the statement.

For example, upper bounds for a conditional statement would be computed according to the rule:

$$u\text{-}bound(\textbf{if } c \textbf{ then } s_1 \textbf{ else } s_2) = u\text{-}bound(c) + max\{u\text{-}bound(s_1), u\text{-}bound(s_2)\}$$

Execution times for individual instructions were assumed to be constant and available from a table in the manual of the processor.

* Work reported herein is supported by the Transregional Collaborative Research Center AVACS of the Deutsche Forschungsgemeinschaft and by the European Network of Excellence ARTIST2.

F.S. de Boer et al. (Eds.): FMCO 2004, LNCS 3657, pp. 317–323, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The above mentioned architectural features introduce "local non-determinism" into the processor behavior; local inspection of the program can not determine the contribution of an instruction to the program's overall execution time. The execution history determines this contribution. It depends on whether in the actual state the instruction's memory accesses hit or miss the cache, whether the pipeline units needed by the instruction are occupied or not, and whether branch prediction succeeds or fails. The variability of an instruction's execution time currently is roughly two orders of magnitude, with an increasing tendency.

This variability of execution times exists on all levels of granularity [TW04], not only for the individual memory access or the single instruction, but also for a context switch, for a function call, for a task or a distributed system of tasks, the communication of a message over a channel, and the delivery of a requested service on top of some middleware.

Lack of information about a system's execution time results from uncertainty of the system environment (input data, timing of input events) and from the interference on shared resources. For example, the variation in execution times for individual instructions results from the competition between different memory accesses and instructions for the caches and for functional units. The variation of communication times is caused by competition for communication channels with restricted bandwidth. This variance is at the heart of non-predictability, since the safe strategy to deal with uncertainty is to assume worst cases and thus overestimate real execution times. Overall, the remedy is over-provisioning.

1.1 Timing Analysis

Methods have been developed and tools based on them have been implemented that bound the variance of instructions' execution times [FHL+01, Wil05]. State-

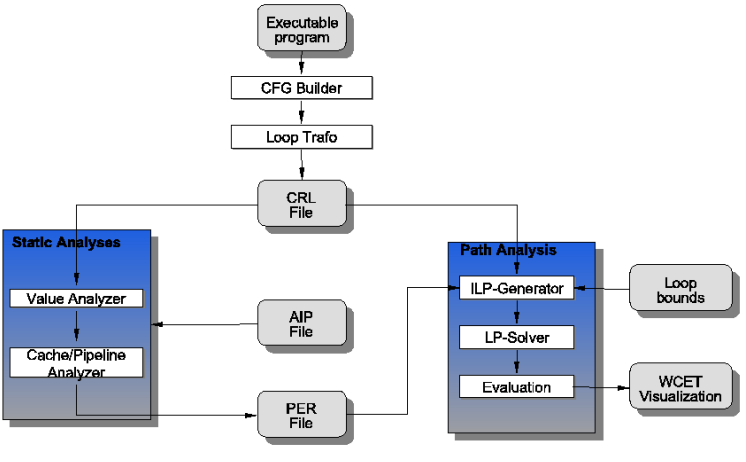


Fig. 1. Architecture of a Timing Analysis tool

of-the-art Timing-Analysis methods split the task into a sequence of subtasks, starting with a number of static analyses, which are based on the theory of Abstract Intepretation [CC77]. The first attempt to determine properties of each task's control flow and the effective addresses of its memory accesses. They use a variant of interval analysis [CH78] to determine the contents in processor registers and the values of variables. These analyses are followed by another one attempting to predict each task's behavior on the processor components such as caches and pipelines. Result of this phase are upper bounds on the execution times of basic blocks. The control flow of each task is translated into an integer linear program with the execution time of the program over all paths as objective function. Maximizing this objective function determines a worst-case path [LMW99, TFW00]. A more or less standard tool architecture has evolved shown in Figure 1.

2 Cost of Uncertainty

Software systems of realistic sizes to be run on powerful processor architectures exhibit state spaces that are too big to be exhaustively analyzed. That's why the above listed sequence of static program analyses use abstraction to reduce this space. The analyses compute at each program point invariants in the form of over-approximations of the set of execution states that can be reached when program execution reaches this point. In general, several invariants are computed for a program point, one for each *context*, i.e., control flow path by which this point can be reached. Differentiating by contexts is absolutely mandatory to obtain enough precision. Each invariant expresses the computed information about the processor's components, e.g. contents of caches, occupancy of pipeline units, state of the branch predictor etc. This information is then used to exclude the possibility of cache misses, pipeline stalls etc. and thereby safely reducing the assumptions about the execution times of instructions.

The information contained in the invariants is by necessity incomplete. First, information about the program's interference with the environment is not available, which may influence the program's execution. This requires that information on several possible control-flow paths be merged. Second, the analyses are based on an abstract model of the processor. This must be conservative with respect to the timing behavior of the concrete processor, but may abstract from details. Third, language features and software design methods can build unsurmountable barriers for even the most powerful analyses as we will attempt to show next.

Figure 2 shows the basic notions we are dealing with. Firstly, the program exhibits a variability in execution time depending on input, thus may have a range of execution times between a best-case and a worst case execution time. However, these two extreme cases can in general not be determined. With the methods described above, one can determine safe lower and upper bounds. Any worst-case guarantee can only be an upper bound. Useful bounds are not too far away from the best-case and worst-case times, resp. A system exhibiting predictable behavior will allow the analyses to arrive at precise bounds.

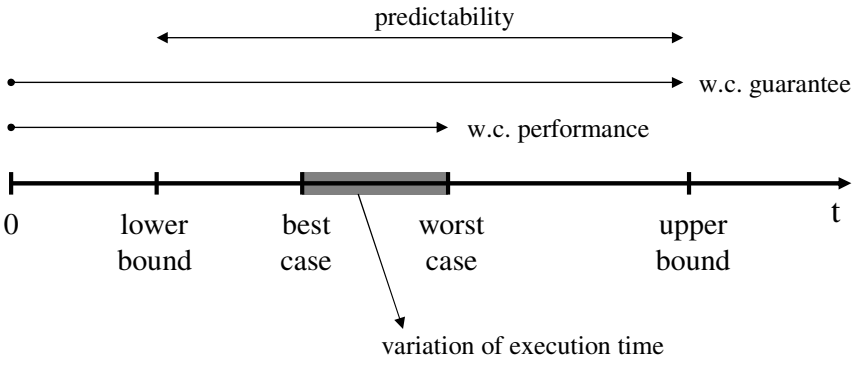


Fig. 2. Basic terms

We will now consider several types of missing information and their costs in terms of precision. Costs are measured in machine cycles. The figures given are taken from a currently popular architecture, a Motorola PowerPC processor equipped with a realistic memory system. A cache analysis will have computed safe, but approximate information about the cache contents at each program point. Missing information about whether an accessed memory block is in the cache has to be accounted for by the cache miss penalty, which is roughly 40 cycles. Depending on the write-back strategy of the cache, we may need to also account for a write back, which means adding another 40 cycles.

Furthermore, assume that the clever designer decided to use virtual memory. This comes with a translation-lookaside buffer (TLB). A TLB miss requires 12 reads and 1 write. Assuming that one can not safely exclude that these reads and writes miss the cache. This means that a TLB miss that can not safely be excluded has to be accounted for with at least $13 \times 40 = 520$ cycles. A page fault costs around 2000 cycles.

Assume, that an object-oriented language has been used for the implementation of the system's tasks. Dynamic method invocation uses a data structure to identify the actual method to activate. An efficient implementation spending some memory overhead for virtual-function tables [WM97] still needs two indirect references to activate the correct method. If we can not exclude the possibility that these two table lookups cause page faults, a method invocation costs 2×2000 cycles in the worst-case.

Now come the *second-order effects*. Let us consider a pointer to data whose value can not be statically inferred. When analyzing an access through this pointer, the analysis must assume accesses to all sets of the cache. With an access to a known address, the cache analysis removes some information, namely the memory block that may be replaced, but it also gains some information, namely the memory block that was loaded into the cache. With unknown access, the analysis only looses information; it must reflect, that one memory block may be removed from each set of the cache, but does not know where the memory block moves into the cache. This is a second-order effect, because it ruins the

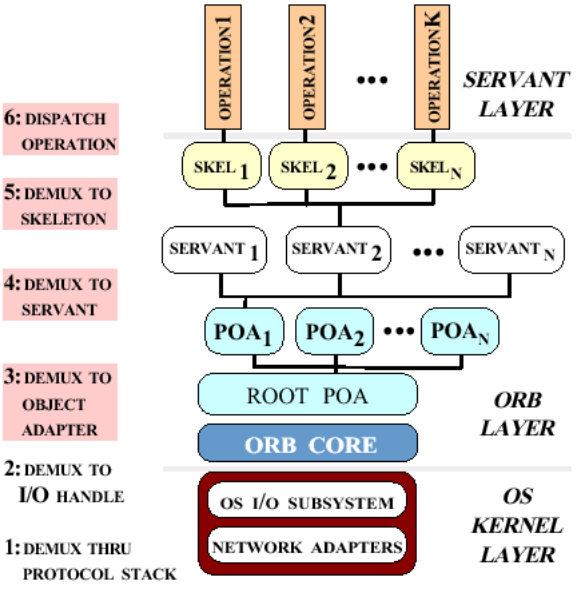


Fig. 3. The ZEN open-source RT-CORBA middleware. Picture taken from [KKSC03]

information about cache contents for future accesses. Even worse are statically unresolvable function pointers. Their damaging effect is of even higher order, because for each indirect call through such a function pointer the worst-case damage for all the potentially called functions has to be assumed.

Middleware has its motivation in the potential for reuse of software components. Object request brokers have to be tailored for the use in real-time systems. In CORBA, requests for services may be served remotely over the net. No time guarantees can be given in this case. A real-time version of CORBA called RT CORBA has been developed. Attempts have been undertaken to increase its predictability [KKSC03]. Figure 3 shows the demultiplexing steps in CORBA request processing. Demultiplexing uses a recursive data structure. To achieve predictability, recursion depth of this data structure has been statically bounded. Still, traversing it to demultiplex a service request potentially causes several page faults and cache misses at very high costs in the case of uncertainty about the memory state.

3 On the Multiplicative Nature of Uncertainty in Layered Systems

Real-life systems are not monolithic, but mostly structured into a layered hierarchy. Often, several layers interfere on a shared resource increasing the variability of execution times. We have already seen, how the brokerage of a service by a

middleware may cause page several faults. The page fault may cause a TLB miss, which in turn may cause several cache misses. Variabilities on different layers combine in a multiplicative way.

The sense behind all this is the following observation:

Observation 1. *At each level, uncertainty has to be accounted for in terms of a number of steps of lower levels. Let us assume that a step on level n costs* m_n^{n-1} *steps of level* $n-1$*. Then it costs* $\prod_{1 \leq i \leq n-1} m_{i-1}^i$ *machine cycles.*

Hence, in the worst case, a negative amplification of mechanisms on different layers will happen.

It should however be stressed that many modern powerful processors exhibit timing anomalies [LS99], which relate execution times on the different levels in more complex ways than the observation indicates. Timing anomalies are counter-intuitive dependencies of a program's execution times on the execution times of individual instructions. Faster execution of an instruction can lead to a longer execution time of the program, and slower execution of an instruction to shorter time for the program. Timing anomalies often are caused by cyclic influences between system components. One such dependency is the following: the contents of the instruction cache determines whether instruction fetch hits or misses the cache, a cache miss may prevent a branch prediction, a wrong branch prediction may ruin the instruction-cache contents. The resulting timing anomaly is, that the local worst case, a cache miss, prevents a branch misprediction, which would have caused a greater damage than the cache miss. Thus, the program runs faster in the case of the cache miss.

4 Towards a Rational Basis for Design

A promising approach to increase predictability would encompass all system layers. It would start with a multiplicative term of the kind given in Observation 1. As stated above, the design would have to exclude timing anomalies or bound the damage on cycles of dependencies. The design would then soften this multiplicative rule by reducing the factors on some layers, if it could make these layers behave predictably by static implementation decisions. This is a successful strategy to achieve a synergistic, quantifiable reduction of variability.

I admit that this looks like a dream in the light of the fact that the trends in systems development go towards less and less predictable systems. Experience with industrial projects lead me to believe that a discipline *Design for Predictability* is needed to reverse these trends and arrive at systems whose behavior as far as consumption of time, space, and energy is predictable.

Acknowledgements

Joint work on increasing the predictability of real-time systems is enjoyed with Lothar Thiele. Contributions to the discussion have come from Stephan Thesing, Oleg Parshin, Reinhold Heckmann, and Peter Marwedel.

References

[CC77] Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Generalized Type Unions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, volume 12(3), pages 77–94, Raleigh, NC, March 1977.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[FHL+01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT*, volume 2211 of *LNCS*, pages 469 – 485, 2001.

[KKSC03] Arvind S. Krishna, Raymond Klefstad, Douglas C. Schmidt, and Angelo Corsaro. Towards predictable real-time java object request brokers. In *Real Time Technology and Applications Symposium*, pages 49–. IEEE, 2003.

[LMW99] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. *Design Automation of Electronic Systems*, 4(3):257–279, 1999.

[LS99] Thomas Lundquist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *20th IEEE Real-Time Systems Symposium*, 1999.

[Sha89] Alan C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.

[TFW00] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Separate Cache and Path Analyses. *Real-Time Systems*, 18(2/3):157–179, May 2000.

[TW04] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28:157 – 177, 2004.

[Wil05] Reinhard Wilhelm. Determination of bounds on execution times. In Richard Zurawski, editor, *Embedded Systems Handbook*, pages 14–1,14–24. CRC Press, 2005.

[WM97] Reinhard Wilhelm and Dieter Maurer. *Compiler Design*. Addison Wesley, 1997.

Author Index

Ábrahám, Erika 49, 296
Albin-Amiot, Hervé 70
Ancona, Davide, 222

Ball, Thomas, 1
Barbosa, Luís S. 23
Behrmann, Gerd 162
Bonsangue, Marcello M. 49, 296

Chatterjee, Krishnendu 141
Cointe, Pierre 70

de Boer, Frank S. 49, 296
De Nicola, Rocco 95
Denier, Simon 70
Di Pierro, Alessandra 120

Grüner, Andreas 296

Hankin, Chris 120
Henzinger, Thomas A. 141

Jifeng, He 183
Johnsen, Einar Broch 274
Jurdziński, Marcin 141

Larsen, Kim G. 162
Li, Xiaoshan 183
Liu, Zhiming 183
Loreti, Michele 95

Moggi, Eugenio 222

Naumann, David A. 251

Owe, Olaf 274

Rasmussen, Jacob I. 162

Steffen, Martin 49, 296

Wiklicky, Herbert 120
Wilhelm, Reinhard 317