

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

LNCS 4709

Formal Methods for Components and Objects

5th International Symposium, FMCO 2006
Amsterdam, The Netherlands, November 2006
Revised Lectures

 Springer

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

University of California, Los Angeles, CA, 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

5th International Symposium, FMCO 2006
Amsterdam, The Netherlands, November 7-10, 2006
Revised Lectures

Volume Editors

Frank S. de Boer
Centre for Mathematics and Computer Science, CWI
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
2300 RA Leiden, The Netherlands
E-mail: marcello@liacs.nl

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

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

Library of Congress Control Number: Applied for

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-74791-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-74791-8 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

springer.com

© Springer-Verlag Berlin Heidelberg 2007
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12119530 06/3180 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, modeling and validation techniques supporting the concepts of reusability and modifiability, and their implementation in new extensions of existing programming languages like Java.

The fifth international symposium on Formal Methods for Components and Objects (FMCO 2006) was held in Amsterdam, The Netherlands, June 7–11, 2007. The program consisted of invited keynote lectures and tutorial lectures selected through a corresponding open-call. The latter provide a tutorial perspective on recent developments. In contrast to many existing conferences, about half of the program consisted of invited keynote lectures by top researchers sharing their interest in the application or development of formal methods for large-scale software systems (object or component oriented). FMCO does not focus on specific aspects of the use of formal methods, but rather it aims at a systematic and comprehensive account of the expanding body of knowledge on modern software systems.

This volume contains the contributions submitted after the symposium by both the invited and selected lecturers. The proceedings of FMCO 2002, FMCO 2003, FMCO 2004 and FMCO 2005 have already been published as volumes 2852, 3188, 3657, and 4111 of Springer's *Lecture Notes in Computer Science*. We believe that these proceedings provide a unique combination of ideas on software engineering and formal methods which reflect the expanding body of knowledge on modern software systems.

Finally, we thank all authors for the high quality of their contributions, and the reviewers for their help in improving the papers for this volume.

June 2007

Frank de Boer
Marcello Bonsangue
Susanne Graf
Willem-Paul de Roever

Organization

The FMCO symposia are organized in the context of the project Mobi-J, 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.

Sponsoring Institutions

The Dutch Organization for Scientific Research (NWO)

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), The Netherlands

The Leiden Institute of Advanced Computer Science (LIACS), The Netherlands

Table of Contents

Testing

Model-Based Testing of Environmental Conformance of Components ...	1
<i>Lars Frantzen and Jan Tretmans</i>	
Exhaustive Testing of Exception Handlers with Enforcer	26
<i>Cyrille Artho, Armin Biere, and Shinichi Honiden</i>	
Model-Based Test Selection for Infinite-State Reactive Systems	47
<i>Bertrand Jeannot, Thierry Jéron, and Vlad Rusu</i>	

Program Verification

Verifying Object-Oriented Programs with KeY: A Tutorial	70
<i>Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Philipp Rümmer, and Peter H. Schmitt</i>	
Rebeca: Theory, Applications, and Tools	102
<i>Marjan Sirjani</i>	
Learning Meets Verification	127
<i>Martin Leucker</i>	

Trust and Security

JACK—A Tool for Validation of Security and Behaviour of Java Applications	152
<i>Gilles Barthe, Lilian Burdy, Julien Charles, Benjamin Grégoire, Marieke Huisman, Jean-Louis Lanet, Mariela Pavlova, and Antoine Requet</i>	
Towards a Formal Framework for Computational Trust	175
<i>Vladimiro Sassone, Karl Krukow, and Mogens Nielsen</i>	

Models of Computation

On Recursion, Replication and Scope Mechanisms in Process Calculi ...	185
<i>Jesús Aranda, Cinzia Di Giusto, Catuscia Palamidessi, and Frank D. Valencia</i>	
Bounded Session Types for Object Oriented Languages	207
<i>Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou, and Nobuko Yoshida</i>	

Distributed Programming

Reflecting on Aspect-Oriented Programming, Metaprogramming, and Adaptive Distributed Monitoring.....	246
<i>Bill Donkervoet and Gul Agha</i>	
Links: Web Programming Without Tiers.....	266
<i>Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop</i>	
Author Index	297

Model-Based Testing of Environmental Conformance of Components

Lars Frantzen^{1,2} and Jan Tretmans^{2,3}

¹ Istituto di Scienza e Tecnologie della Informazione “Alessandro Faedo”
Consiglio Nazionale delle Ricerche, Pisa – Italy

`lars.frantzen@isti.cnr.it`

² Institute for Computing and Information Sciences
Radboud University Nijmegen – The Netherlands

`{lf,tretmans}@cs.ru.nl`

³ Embedded Systems Institute
Eindhoven – The Netherlands

`jan.tretmans@esi.nl`

Abstract. In component-based development, the correctness of a system depends on the correctness of the individual components and on their interactions. Model-based testing is a way of checking the correctness of a component by means of executing test cases that are systematically generated from a model of the component. This model should include the behaviour of how the component can be invoked, as well as how the component itself invokes other components. In many situations, however, only a model that specifies how others can use the component, is available. In this paper we present an approach for model-based testing of components where only these available models are used. Test cases for testing whether a component correctly reacts to invocations are generated from this model, whereas the test cases for testing whether a component correctly invokes other components, are generated from the models of these other components. A formal elaboration is given in the realm of labelled transition systems. This includes an implementation relation, called **eco**, which formally defines when a component is correct with respect to the components it uses, and a sound and exhaustive test generation algorithm for **eco**.

1 Introduction

Software testing involves checking of desired properties of a software product by systematically executing the software, while stimulating it with test inputs, and observing and checking the execution results. Testing is a widely used technique to assess the quality of software, but it is also a difficult, error-prone, and labor-intensive technique. Consequently, test automation is an important area of research and development: without automation it will not be feasible to test future generations of software products in an effective and efficient manner. Automation of the testing process involves automation of the execution of test cases, automation of the analysis of test results, as well as automation of the generation of sufficiently many and valid test cases.

Model-Based Testing. One of the emerging and promising techniques for test automation is model-based testing. In model based testing, a *model* of the desired behavior of the *implementation under test* (IUT) is the starting point for test generation and serves as the oracle for test result analysis. Large amounts of test cases can, in principle, be algorithmically and completely automatically generated from the model. If this model is valid, i.e., expresses precisely what the implementation under test should do, all these tests are valid, too. Model-based testing has recently gained increased attention with the popularization of modeling itself.

Most model-based testing methods deal with black-box testing of functionality. This implies that the kind of properties being tested concern the functionality of the system. Functionality properties express whether the system correctly does what it should do in terms of correct responses to given stimuli, as opposed to, e.g., performance, usability, or reliability properties. In black-box testing, the specification is the starting point for testing. The specification prescribes what the IUT should do, and what it should not do, in terms of the behavior observable at its external interfaces. The IUT is seen as a black box without internal detail, as opposed to white-box testing, where the internal structure of the IUT, i.e., the program code, is the basis for testing. Also in this paper we will restrict ourselves to black-box testing of functionality properties.

Model-based testing with labelled transition systems. One of the formal theories for model-based testing uses labelled transition systems as models, and a formal implementation relation called **ioco** for defining conformance between an IUT and a specification [10,11]. A labelled transition system is a structure with states representing the states of the system, and with transitions between states representing the actions that the system may perform. The implementation relation **ioco** expresses that an IUT conforms to its specification if the IUT never produces an output that cannot be produced by the specification. In this theory, an algorithm for the generation of test cases exists, which is provably sound for **ioco**-conformance, i.e., generated test cases only detect **ioco** errors, and exhaustive, i.e., all potential **ioco** errors can be detected.

Testing of Components. In component-based development, systems are built by gluing components together. Components are developed separately, often by different manufacturers, and they can be reused in different environments. A component is responsible for performing a specific task, or for delivering a specified service. A user requesting this service will invoke the component to provide its service. In doing so, the component may, in turn, invoke other components for providing their services, and these invoked components may again use other components. A component may at the same time act as a service provider and as a service requester.

A developer who composes a system from separate components, will only know about the services that the components perform, and not about their internal details. Consequently, clear and well-specified interfaces play a crucial role in

component technology, and components shall correctly implement these interface specifications. Correctness involves both the component's role as a service provider and its role as a service requester: a component must correctly provide its specified service, as well as correctly use other components.

Component-based testing. In our black-box setting, component-based testing concerns testing of behavior as it is observed at the component's interfaces. This applies to testing of individual components as well as to testing of aggregate systems built from components, and it applies to testing of provided services, as well as to testing of how other services are invoked.

When testing aggregated systems this can be done "bottom-up", i.e., starting with testing the components that do not invoke other components, and then adding components to the system that use the components already tested, and so forth, until the highest level has been reached. Another approach is to use *stubs* to simulate components that are invoked, so that a component can be tested without having the components available that are invoked by the component under test.

Model-based testing of components. For model-based testing of an individual component, we, in principle, need a complete model of the component. Such a model should specify the behavior at the service providing interface, the behavior at the service requesting interface, and the mutual dependencies between actions at both interfaces. Such a complete model, however, is often not available. Specifications of components are usually restricted to the behavior of the provided services. The specification of how other components are invoked is considered an internal implementation detail, and, from the point of view of a user of an aggregate system, it is.

Goal. The aim of this paper is to present an approach for model-based testing of a component at both the service providing interface and the requesting interface in a situation where a complete behavior model is not available. The approach assumes that a specification of the provided service is available for both the component under test, and for the components being invoked by the component under test. Test cases for the provided service are derived from the corresponding service specification. Test cases for checking how the component requests services from other components are derived from the provided service specifications of these other components.

The paper builds on the **io**co-test theory for labelled transition systems, it discusses where this theory is applicable for testing components, and where it is not. A new implementation relation is introduced called *environmental conformance* – **eco**. This relation expresses that a component correctly invokes another component according to the provided service specification of that other component. A complete (sound and exhaustive) test generation algorithm for **eco** is given.

Overview. Section 2 starts with recalling the most important concepts of the **io**co-test theory for labelled transition systems, after which Section 3 sets the

scene for formally testing components. The implementation relation **eco** is introduced in Section 4, followed by the test generation algorithm in Section 5. The combination of testing at different interfaces is briefly discussed in Section 6. Concluding remarks are presented in Section 7.

2 Testing for Labelled Transition Systems

Model-based testing deals with models, correctness (or conformance-) relations, test cases, test generation algorithms, and soundness and exhaustiveness of the generated test cases with respect to the conformance relations. This section presents the formal test theory for labelled transition systems using the **ico**-conformance relation; see [10,11]. This theory will be our starting point for the discussion of model-based testing of components in the next sections.

Models. In the **ico**-test theory, formal specifications, implementations, and test cases are all expressed as labelled transition systems.

Definition 1. A labelled transition system with inputs and outputs is a 5-tuple $\langle Q, L_I, L_U, T, q_0 \rangle$ where Q is a countable, non-empty set of states; L_I is a countable set of input labels; L_U is a countable set of output labels, such that $L_I \cap L_U = \emptyset$; $T \subseteq Q \times (L_I \cup L_U \cup \{\tau\}) \times Q$, with $\tau \notin L_I \cup L_U$, is the transition relation; and $q_0 \in Q$ is the initial state.

The labels in L_I and L_U represent the inputs and outputs, respectively, of a system, i.e., the system's possible interactions with its environment¹. Inputs are usually decorated with '?' and outputs with '!'. We use $L = L_I \cup L_U$ when we abstract from the distinction between inputs and outputs.

The execution of an action is modeled as a transition: $(q, \mu, q') \in T$ expresses that the system, when in state q , may perform action μ , and go to state q' . This is more elegantly denoted as $q \xrightarrow{\mu} q'$. Transitions can be composed: $q \xrightarrow{\mu} q' \xrightarrow{\mu'} q''$, which is written as $q \xrightarrow{\mu \cdot \mu'} q''$.

Internal transitions are labelled by the special action τ ($\tau \notin L$), which is assumed to be unobservable for the system's environment. Consequently, the observable behavior of a system is captured by the system's ability to perform sequences of observable actions. Such a sequence of observable actions, say σ , is obtained from a sequence of actions under abstraction from the internal action τ , and it is denoted by $\xrightarrow{\sigma}$. If, for example, $q \xrightarrow{a \cdot \tau \cdot \tau \cdot b \cdot c \cdot \tau} q'$ ($a, b, c \in L$), then we write $q \xrightarrow{a \cdot b \cdot c} q'$ for the τ -abstracted sequence of observable actions. We say that q is able to perform the *trace* $a \cdot b \cdot c \in L^*$. Here, the set of all finite sequences over L is denoted by L^* , with ϵ denoting the empty sequence. If $\sigma_1, \sigma_2 \in L^*$ are finite sequences, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2 . Some more, standard notations and definitions are given in Definitions 2 and 3.

¹ The 'U' refers to 'uitvoer', the Dutch word for 'output', which is preferred for historical reasons, and to avoid confusion between L_O (letter 'O') and L_0 (digit zero).

Definition 2. Let $p = \langle Q, L_I, L_U, T, q_0 \rangle$ be a labelled transition system with $q, q' \in Q$, $\mu, \mu_i \in L \cup \{\tau\}$, $a, a_i \in L$, and $\sigma \in L^*$.

$$\begin{array}{ll}
q \xrightarrow{\mu} q' & \Leftrightarrow_{\text{def}} (q, \mu, q') \in T \\
q \xrightarrow{\mu_1 \cdots \mu_n} q' & \Leftrightarrow_{\text{def}} \exists q_0, \dots, q_n : q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \dots \xrightarrow{\mu_n} q_n = q' \\
q \xrightarrow{\mu_1 \cdots \mu_n} & \Leftrightarrow_{\text{def}} \exists q' : q \xrightarrow{\mu_1 \cdots \mu_n} q' \\
q \xrightarrow{\mu_1 \cdots \mu_n} / & \Leftrightarrow_{\text{def}} \text{not } \exists q' : q \xrightarrow{\mu_1 \cdots \mu_n} q' \\
q \xrightarrow{\epsilon} q' & \Leftrightarrow_{\text{def}} q = q' \text{ or } q \xrightarrow{\tau \cdots \tau} q' \\
q \xrightarrow{a} q' & \Leftrightarrow_{\text{def}} \exists q_1, q_2 : q \xrightarrow{\epsilon} q_1 \xrightarrow{a} q_2 \xrightarrow{\epsilon} q' \\
q \xrightarrow{a_1 \cdots a_n} q' & \Leftrightarrow_{\text{def}} \exists q_0 \dots q_n : q = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n = q' \\
q \xrightarrow{\sigma} & \Leftrightarrow_{\text{def}} \exists q' : q \xrightarrow{\sigma} q' \\
q \not\xrightarrow{\sigma} & \Leftrightarrow_{\text{def}} \text{not } \exists q' : q \xrightarrow{\sigma} q'
\end{array}$$

In our reasoning about labelled transition systems we will not always distinguish between a transition system and its initial state. If $p = \langle Q, L_I, L_U, T, q_0 \rangle$, we will identify the process p with its initial state q_0 , and, e.g., we write $p \xrightarrow{\sigma}$ instead of $q_0 \xrightarrow{\sigma}$.

Definition 3. Let p be a (state of a) labelled transition system, P a set of states, $A \subseteq L$ a set of labels, and $\sigma \in L^*$.

1. $\text{traces}(p) =_{\text{def}} \{ \sigma \in L^* \mid p \xrightarrow{\sigma} \}$
2. $p \text{ after } \sigma =_{\text{def}} \{ p' \mid p \xrightarrow{\sigma} p' \}$
3. $P \text{ after } \sigma =_{\text{def}} \bigcup \{ p \text{ after } \sigma \mid p \in P \}$
4. $P \text{ refuses } A =_{\text{def}} \exists p \in P, \forall \mu \in A \cup \{\tau\} : p \not\xrightarrow{\mu}$

The class of labelled transition systems with inputs in L_I and outputs in L_U is denoted as $\mathcal{LTS}(L_I, L_U)$. For technical reasons we restrict this class to *strongly converging* and *image finite* systems. Strong convergence means that infinite sequences of τ -actions are not allowed to occur. Image finiteness means that the number of non-deterministically reachable states shall be finite, i.e., for any σ , $p \text{ after } \sigma$ shall be finite.

Representing labelled transition systems. To represent labelled transition systems we use either graphs (as in Fig. [1](#)), or expressions in a process-algebraic-like language with the following syntax:

$$B ::= a ; B \mid \mathbf{i} ; B \mid \Sigma B \mid B \parallel [G] B \mid P$$

Expressions in this language are called behavior expressions, and they define labelled transition systems following the axioms and rules given in Table [1](#).

In that table, $a \in L$ is a label, B is a behavior expression, \mathcal{B} is a countable set of behavior expressions, $G \subseteq L$ is a set of labels, and P is a *process name*, which must be linked to a named behavior expression by a process definition of the form $P := B_P$. In addition, we use $B_1 \square B_2$ as an abbreviation for $\Sigma \{B_1, B_2\}$, **stop** to denote $\Sigma \emptyset$, \parallel as an abbreviation for $\parallel [L]$, i.e., synchronization on all observable actions, and $\parallel\parallel$ as an abbreviation for $\parallel [\emptyset]$, i.e., full interleaving without synchronization.

Table 1. Structural operational semantics

$$\begin{array}{c}
\frac{}{a;B \xrightarrow{a} B} \quad \frac{}{\mathbf{i};B \xrightarrow{\tau} B} \quad \frac{B \xrightarrow{\mu} B'}{\Sigma B \xrightarrow{\mu} B'} \quad B \in \mathcal{B}, \mu \in L \cup \{\tau\} \\
\\
\frac{B_1 \xrightarrow{\mu} B'_1}{B_1 \llbracket G \rrbracket B_2 \xrightarrow{\mu} B'_1 \llbracket G \rrbracket B_2} \quad \frac{B_2 \xrightarrow{\mu} B'_2}{B_1 \llbracket G \rrbracket B_2 \xrightarrow{\mu} B_1 \llbracket G \rrbracket B'_2} \quad \mu \in (L \cup \{\tau\}) \setminus G \\
\\
\frac{B_1 \xrightarrow{a} B'_1, B_2 \xrightarrow{a} B'_2}{B_1 \llbracket G \rrbracket B_2 \xrightarrow{a} B'_1 \llbracket G \rrbracket B'_2} \quad a \in G \quad \frac{B_P \xrightarrow{\mu} B'}{P \xrightarrow{\mu} B'} \quad P := B_P, \mu \in L \cup \{\tau\}
\end{array}$$

Input-output transition systems. In model-based testing there is a specification, which prescribes what an IUT shall do, and there is the IUT itself which is a black-box performing some behavior. In order to formally reason about the IUT's behavior the assumption is made that the IUT behaves as if it were some kind of formal model. This assumption is sometimes referred to as the test assumption or test hypothesis.

In the **io-co**-test theory a specification is a labelled transition system in $\mathcal{LTS}(L_I, L_U)$. An implementation is assumed to behave as if it were a labelled transition system that is always able to perform any input action, i.e., all inputs are enabled in all states. Such a system is defined as an *input-output transition system*. The class of such input-output transition systems is denoted by $\mathcal{IOTS}(L_I, L_U) \subseteq \mathcal{LTS}(L_I, L_U)$.

Definition 4. An input-output transition system is a labelled transition system with inputs and outputs $\langle Q, L_I, L_U, T, q_0 \rangle$ where all input actions are enabled in any reachable state:

$$\forall \sigma, q : q_0 \xrightarrow{\sigma} q \text{ implies } \forall a \in L_I : q \xrightarrow{a}$$

A state of a system where no outputs are enabled, and consequently the system is forced to wait until its environment provides an input, is called *suspended*, or *quiescent*. An observer looking at a quiescent system does not see any outputs. This particular observation of seeing nothing can itself be considered as an event, which is denoted by δ ($\delta \notin L \cup \{\tau\}$); $p \xrightarrow{\delta} p$ expresses that p allows the observation of quiescence. Also these transitions can be composed, e.g., $p \xrightarrow{\delta \cdot ?a \cdot \delta \cdot ?b \cdot !x}$ expresses that initially p is quiescent, i.e., does not produce outputs, but p does accept input action $?a$, after which there are again no outputs; when then input $?b$ is performed, the output $!x$ is produced. We use L_δ for $L \cup \{\delta\}$, and traces that may contain the quiescence action δ are called *suspension traces*.

Definition 5. Let $p = \langle Q, L_I, L_U, T, q_0 \rangle \in \mathcal{LTS}(L_I, L_U)$.

1. A state q of p is quiescent, denoted by $\delta(q)$, if $\forall \mu \in L_U \cup \{\tau\} : q \not\xrightarrow{\mu}$

2. $p_\delta =_{\text{def}} \langle Q, L_I, L_U \cup \{\delta\}, T \cup T_\delta, q_0 \rangle$,
with $T_\delta =_{\text{def}} \{ q \xrightarrow{\delta} q \mid q \in Q, \delta(q) \}$
3. The suspension traces of p are $\text{Straces}(p) =_{\text{def}} \{ \sigma \in L_\delta^* \mid p_\delta \xRightarrow{\sigma} \}$

From now on we will usually include δ -transitions in the transition relations, i.e., we consider p_δ instead of p , unless otherwise indicated. Definitions 2 and 3 also apply to transition systems with label set L_δ .

*The implementation relation **io**co.* An implementation relation is intended to precisely define when an implementation is correct with respect to a specification. The first implementation relation that we consider is **io**co, which is abbreviated from **input-output conformance**. Informally, an implementation $i \in \mathcal{IOTS}(L_I, L_U)$ is **io**co-conforming to specification $s \in \mathcal{LTS}(L_I, L_U)$ if any experiment derived from s and executed on i leads to an output (including quiescence) from i that is foreseen by s . We define **io**co as a special case of the more general class of relations **io**co \mathcal{F} , where $\mathcal{F} \subseteq L_\delta^*$ is a set of suspension traces, which typically depends on the specification s .

Definition 6. Let q be a state in a transition system, Q be a set of states, $i \in \mathcal{IOTS}(L_I, L_U)$, $s \in \mathcal{LTS}(L_I, L_U)$, and $\mathcal{F} \subseteq (L_I \cup L_U \cup \{\delta\})^*$, then

1. $\text{out}(q) =_{\text{def}} \{ x \in L_U \mid q \xrightarrow{x} \} \cup \{ \delta \mid \delta(q) \}$
2. $\text{out}(Q) =_{\text{def}} \bigcup \{ \text{out}(q) \mid q \in Q \}$
3. $i \text{ io}_{\mathcal{F}} s \Leftrightarrow_{\text{def}} \forall \sigma \in \mathcal{F} : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)$
4. $i \text{ io} s \Leftrightarrow_{\text{def}} i \text{ io}_{\text{Straces}(s)} s$

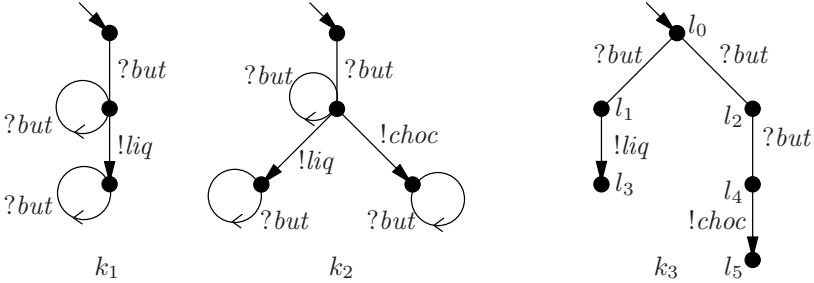


Fig. 1. Example labelled transition systems

Example 1. Figure 1 presents three examples of labelled transition systems modeling candy machines. There is an input action for pushing a button $?but$, and there are outputs for obtaining chocolate $!choc$ and liquorice $!liq$: $L_I = \{?but\}$ and $L_U = \{!liq, !choc\}$.

Since $k_1, k_2 \in \mathcal{IOTS}(L_I, L_U)$ they can be both specifications and implementations; k_3 is not input-enabled, and can only be a specification. We have that $\text{out}(k_1 \text{ after } ?but) = \{!liq\} \subseteq \{!liq, !choc\} = \text{out}(k_2 \text{ after } ?but)$; so we get now

k_1 **io**co k_2 , but k_2 **io**co k_1 . For k_3 we have $out(k_3 \text{ after } ?but) = \{!liq, \delta\}$ and $out(k_3 \text{ after } ?but \cdot ?but) = \{!choc\}$, so both k_1, k_2 **io**co k_3 .

The importance of having suspension actions δ in the set \mathcal{F} over which **io**co quantifies is also illustrated in Fig. 2. It holds that $out(r_1 \text{ after } ?but \cdot ?but) = out(r_2 \text{ after } ?but \cdot ?but) = \{!liq, !choc\}$, but we have $out(r_1 \text{ after } ?but \cdot \delta \cdot ?but) = \{!liq, !choc\} \supset \{!choc\} = out(r_2 \text{ after } ?but \cdot \delta \cdot ?but)$. So, without δ in these traces r_1 and r_2 would be considered implementations of each other in both directions, whereas with δ , r_2 **io**co r_1 but r_1 **io**co r_2 .

Underspecification and the implementation relation uioco. The implementation relation **io**co allows to have partial specifications. A partial specification does not specify the required behavior of the implementation after all possible traces. This corresponds to the fact that specifications may be non-input enabled, and inclusion of *out*-sets is only required for suspension traces that explicitly occur in the specification. Traces that do not explicitly occur are called underspecified. There are different ways of dealing with underspecified traces. The relation **uioco** does it in a slightly different manner than **io**co. For the rationale consider Example 2.

Example 2. Consider k_3 of Fig. 1 as a specification. Since k_3 is not input-enabled, it is a partial specification. For example, $?but \cdot ?but \cdot ?but$ is an underspecified trace, and any implementation behavior is allowed after it. On the other hand, $?but$ is clearly specified; the allowed outputs after it are $!liq$ and δ . For the trace $?but \cdot ?but$ the situation is less clear. According to **io**co the expected output after $?but \cdot ?but$ is $out(k_3 \text{ after } ?but \cdot ?but) = \{!choc\}$. But suppose that in the first $?but$ -transition k_3 moves nondeterministically to state l_1 (the left branch) then one might argue that the second $?but$ -transition is underspecified, and that, consequently, any possible behavior is allowed in an implementation. This is exactly where **io**co and **uioco** differ: **io**co postulates that $?but \cdot ?but$ is not an underspecified trace, because there exists a state where it is specified, whereas **uioco** states that $?but \cdot ?but$ is underspecified, because there exists a state where it is underspecified.

Formally, **io**co quantifies over $\mathcal{F} = Straces(s)$, which are all possible suspension traces of the specification s . The relation **uioco** quantifies over $\mathcal{F} = Utraces(s) \subseteq Straces(s)$, which are the suspension traces without the possibly underspecified traces, i.e., all suspension traces σ of s for which it is *not* possible that a prefix σ_1 of σ ($\sigma = \sigma_1 \cdot a \cdot \sigma_2$) leads to a state of s where the remainder $a \cdot \sigma_2$ of σ is underspecified, that is, a is refused.

Definition 7. Let $i \in \mathcal{IOTS}(L_I, L_U)$, and $s \in \mathcal{LTS}(L_I, L_U)$.

1. $Utraces(s) =_{\text{def}} \{ \sigma \in Straces(s) \mid \forall \sigma_1, \sigma_2 \in L_\delta^*, a \in L_I : \sigma = \sigma_1 \cdot a \cdot \sigma_2 \text{ implies not } s \text{ after } \sigma_1 \text{ refuses } \{a\} \}$
2. $i \text{ uioco } s \Leftrightarrow_{\text{def}} i \text{ io}co_{Utraces(s)} s$

Example 3. Because $Utraces(s) \subseteq Straces(s)$ it is evident that **uioco** is not stronger than **io**co. That it is strictly weaker follows from the following example. Take k_3 in Fig. 1 as a (partial) specification, and consider r_1 and r_2 from Fig. 2 as potential implementations. Then r_2 **io**co k_3 because $!liq \in out(r_2 \text{ after } ?but \cdot ?but)$

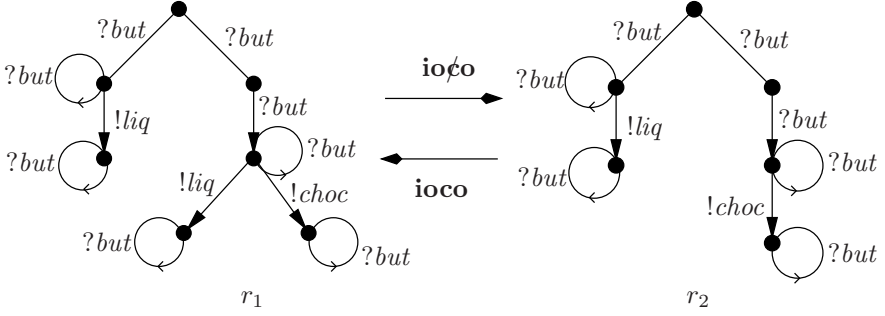


Fig. 2. More labelled transition systems

and $!liq \notin out(k_3 \text{ after } ?but \cdot ?but)$. But $r_2 \mathbf{uio} k_3$ because we have $?but \cdot ?but \notin Utraces(k_3)$. Also $r_1 \mathbf{ioco} k_3$, but in this case also $r_1 \mathbf{uio} k_3$. The reason for this is that we have $?but \cdot \delta \cdot ?but \in Utraces(k_3)$, $!liq \in out(r_1 \text{ after } ?but \cdot \delta \cdot ?but)$ and $!liq \notin out(k_3 \text{ after } ?but \cdot \delta \cdot ?but)$.

Test Cases. For the generation of test cases from labelled transition system specifications, which can test implementations that behave as input-output transition systems, we must first define what test cases are. Then we discuss what test execution is, what it means to pass a test, and which correctness properties should hold for generated test cases so that they will detect all and only non-conforming implementations. A test generation algorithm is not given in this section; for \mathbf{ioco} and \mathbf{uio} test generation algorithms we will refer to other publications. In Sect. 5, this paper will give a test generation algorithm for the new implementation relation \mathbf{eco} for component conformance, which will be defined in Sect. 4.

A test case is a specification of the behavior of a tester in an experiment carried out on an implementation under test. The behavior of such a tester is also modeled as a special kind of input-output transition system, but, naturally, with inputs and outputs exchanged. Consequently, input-enabledness of a test case means that all actions in L_U (i.e., the set of outputs of the implementation) are enabled. For observing quiescence we add a special label θ to the transition systems modeling tests ($\theta \notin L$).

Definition 8. A test case t for an implementation with inputs L_I and outputs L_U is an input-output transition system $\langle Q, L_U, L_I \cup \{\theta\}, T, q_0 \rangle \in IOTS(L_U, L_I \cup \{\theta\})$ generated following the next fragment of the syntax for behavior expressions, where **pass** and **fail** are process names:

$$\begin{aligned}
 t ::= & \mathbf{pass} \\
 & | \mathbf{fail} \\
 & | \Sigma \{ x ; t \mid x \in L_U \cup \{a\} \} \text{ for some } a \in L_I \\
 & | \Sigma \{ x ; t \mid x \in L_U \cup \{\theta\} \} \\
 & \text{where } \mathbf{pass} := \Sigma \{ x ; \mathbf{pass} \mid x \in L_U \cup \{\theta\} \} \\
 & \quad \mathbf{fail} := \Sigma \{ x ; \mathbf{fail} \mid x \in L_U \cup \{\theta\} \}
 \end{aligned}$$

The class of test cases for implementations with inputs L_I and outputs L_U is denoted as $\mathcal{TTS}(L_U, L_I)$. For testing an implementation, normally a set of test cases is used. Such a set is called a *test suite* $T \subseteq \mathcal{TTS}(L_U, L_I)$.

Test Execution. Test cases are run by putting them in parallel with the implementation under test, where inputs of the test case synchronize with the outputs of the implementations, and vice versa. Basically, this can be modeled using the behavior-expression operator \parallel . Since, however, we added the special label θ to test cases to test for quiescence, this operator has to be extended a bit, and is then denoted as $\parallel\!\!\parallel$.

Because of nondeterminism in implementations, it may be the case that testing the same implementation with the same test case may lead to different test results. An implementation passes a test case if and only if all its test runs lead to a pass state of the test case. All this is reflected in the following definition.

Definition 9. Let $t \in \mathcal{TTS}(L_U, L_I)$ and $i \in \mathcal{IOTS}(L_I, L_U)$.

1. Running a test case t with an implementation i is expressed by the parallel operator $\parallel\!\!\parallel : \mathcal{TTS}(L_U, L_I) \times \mathcal{IOTS}(L_I, L_U) \rightarrow \mathcal{LTS}(L_I \cup L_U \cup \{\theta\})$ which is defined by the following inference rules:

$$\frac{i \xrightarrow{\tau} i'}{t \parallel\!\!\parallel i \xrightarrow{\tau} t \parallel\!\!\parallel i'} \quad \frac{t \xrightarrow{a} t', i \xrightarrow{a} i'}{t \parallel\!\!\parallel i \xrightarrow{a} t' \parallel\!\!\parallel i'} \quad a \in L_I \cup L_U \quad \frac{t \xrightarrow{\theta} t', i \xrightarrow{\delta} \delta}{t \parallel\!\!\parallel i \xrightarrow{\theta} t' \parallel\!\!\parallel i}$$

2. A test run of t with i is a trace of $t \parallel\!\!\parallel i$ leading to one of the states **pass** or **fail** of t :

$$\sigma \text{ is a test run of } t \text{ and } i \Leftrightarrow_{\text{def}} \exists i' : t \parallel\!\!\parallel i \xrightarrow{\sigma} \mathbf{pass} \parallel\!\!\parallel i' \text{ or } t \parallel\!\!\parallel i \xrightarrow{\sigma} \mathbf{fail} \parallel\!\!\parallel i'$$

3. Implementation i passes test case t if all test runs go to the **pass**-state of t :

$$i \text{ passes } t \Leftrightarrow_{\text{def}} \forall \sigma \in L_{\theta}^*, \forall i' : t \parallel\!\!\parallel i \not\xrightarrow{\sigma} \mathbf{fail} \parallel\!\!\parallel i'$$

4. An implementation i passes a test suite T if it passes all test cases in T :

$$i \text{ passes } T \Leftrightarrow_{\text{def}} \forall t \in T : i \text{ passes } t$$

If i does not pass a test case or a test suite, it **fails**.

Completeness of testing. For **io**co-testing a couple of algorithms exist that can generate test cases from labelled transition system specifications [10][12][8]. These algorithms have been shown to be correct, in the sense that the test suites generated with these algorithms are able to detect all, and only all, non-**io**co correct implementations. This is expressed by the properties of soundness and exhaustiveness. A test suite is sound if any test run leading to **fail** indicates an error, and a test suite is exhaustive if all possible errors in implementations can be detected. Of course, exhaustiveness is merely a theoretical property: for realistic systems exhaustive test suites would be infinite, both in number of test cases and in the size of test cases. But yet, exhaustiveness does express that there are no **io**co-errors that are undetectable.

Definition 10. Let s be a specification and T a test suite; then for **ioco**:

$$\begin{aligned} T \text{ is sound} & \Leftrightarrow_{\text{def}} \forall i \in \mathcal{IOTS}(L_I, L_U) : i \text{ ioco } s \text{ implies } i \text{ passes } T \\ T \text{ is exhaustive} & \Leftrightarrow_{\text{def}} \forall i \in \mathcal{IOTS}(L_I, L_U) : i \text{ ioco } s \quad \text{if} \quad i \text{ passes } T \end{aligned}$$

3 Towards Formal Component-Based Testing

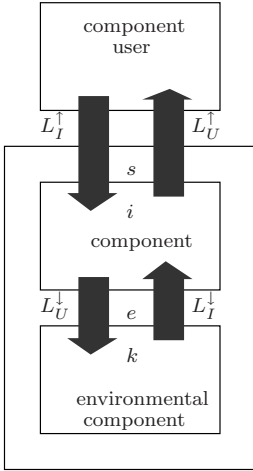
Correctness of components. In component-based testing we wish to test components. A component is a (software) entity that provides some *service* to a potential user. A user can invoke, or request this service. The service is provided via some *interface* of the component, referred to as the *service interface*, *providing interface*, *called interface*, or *upper interface*. A component, in turn, may use other components in its environment, i.e., the component acts as a user of, or requests a service from another component, which, in turn, provides that service. The services provided by these other components are requested via another interface, to which we refer as *required interface*, *calling interface*, or *lower interface*; see Fig. 3(a).

For a service requester it is transparent whether the component i invokes services of other environmental components, like k , at its lower interface, or not. The service requester is only interested whether the component i provides the requested service at the service interface in compliance with its specification s .

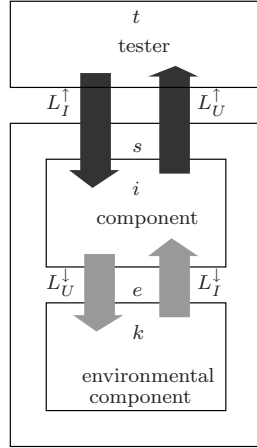
On the other hand, the environmental component k that is being invoked via the lower interface of i , does not care about the service being provided by the component i . It only cares whether the component i correctly requests for the services that the environmental component k provides, according to the rules laid down in k 's service specification e .

Yet, although the correctness requirements on the behavior of a component can be clearly split into requirements on the upper interface and requirements on the lower interface, the correctness of the whole component, naturally, involves correct behavior on both interfaces. Moreover, the behavior of the component on both interfaces is in general not independent: a service request to an environmental component at the lower interface is typically triggered by a service request at the upper interface, and the result of the latter depends on the result of the first.

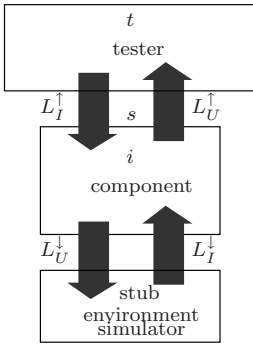
When specifying components, the emphasis is usually on the specification of the provided service, since this is what the component must fulfill and what a user of the component sees. The component's behavior at the lower interface is often not specified. It can only be indirectly derived from what the environmental component expects, i.e., from the provided service specification of that used component. In this paper we will formalize model-based testing of components at their lower interface using the upper interface specification of the environmental component that is invoked. By so doing, we strictly split the requirements on the lower interface from the requirements on the upper interface, since this is the only passable way to go when only specifications of the provided services are available.



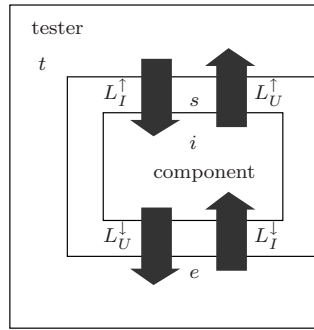
(a) A component i with its "lower-level" environment k .



(b) A composed system of i and k tested at the service level.



(c) A component i tested with a stub at its lower interface.



(d) A component i tested concurrently at the service- and lower interface in a "horse-shoe".

Fig. 3. Component-based testing

This is also the approach of recent, service-oriented testing frameworks like *audition* [2]. This framework assumes behavioral specifications of the provided service interfaces. Based on these specifications, a testing phase is introduced when services ask for being published at a service registry – the service undergoes a monitored trial before being put “on stage”. During this testing phase, the service under test is actively tested at its upper interface, and it is additionally tested, whether the service correctly invokes other services via its lower interface.

If, instead, one wants to take requirements on the interdependency between the interfaces into account, more complete specifications are needed. This is not treated in this paper. For a survey of component-based testing see [9,6].

Formalizing components. We will formalize the behavior of component services in the realm of labelled transition systems. Fig. 3(a) gives a first step towards the formalization of these concepts. The component under consideration is a component implementation denoted by i ; i is an input-output transition system, or, more precisely, the implementation i , which is seen as a black-box, is assumed to behave as an input-output transition system (cf. Section 2: test assumption). The actions that can occur at the upper interface are inputs L_I^\uparrow and outputs L_U^\uparrow , whereas L_I^\downarrow and L_U^\downarrow represent the inputs and outputs, respectively, at the lower interface. Thus $i \in \mathcal{IOTS}(L_I^\uparrow \cup L_I^\downarrow, L_U^\uparrow \cup L_U^\downarrow)$.

The service to be provided by the component at the upper interface is specified by s , which only involves the upper interface: $s \in \mathcal{LTS}(L_I^\uparrow, L_U^\uparrow)$. The behavior of the provided service of the environmental component used by i is specified by $e \in \mathcal{LTS}(L_U^\downarrow, L_I^\downarrow)$, and implemented by $k \in \mathcal{IOTS}(L_U^\downarrow, L_I^\downarrow)$. Only the actions at the lower interface of i , which correspond to the actions of the upper interface of the invoked environmental component k , but with inputs and outputs exchanged, are involved here. Of course, the environmental component, in turn, may have a lower interface via which it will invoke yet other components, but for the component i , being just a service requester for e , this is transparent. In addition, in realistic situations i will usually request services from several different components, but we restrict our discussion to only one service being called. Considering several environmental components can in this setting, for instance, be expressed as their parallel (interleaved) composition, leading again to a single component.

Typically, input actions at the upper interface model the request for, i.e., the start of a service, whereas output actions model the result provided by that service. Conversely, at the lower interface the output actions model requests to an environmental component, whereas input actions model the results provided by the environmental component.

Testing components. A component can be tested in different ways. The simplest, and often used way is to test at the upper interface as in Fig. 3(b). This leads to a "bottom-up" test strategy, where the components that do not invoke other components, are tested first. After this, components are added that use these already tested components, so that these subsystems can be tested, to which then again components can be added, until all components have been added and tested. In principle, this way of testing is sufficient in the sense that all functionality that is observable from a service requester (user) point of view is tested. There are some disadvantages of this testing method, though. The first is that the behavior at the lower interface of the component is not thoroughly tested. This apparently did not lead to failures in the services provided (because these were tested), but it might cause problems when a component is replaced by a new or other (version of the) component, or if a component is reused in another environment. For instance, one environmental component may be robust

enough to deal with certain erroneous invocations, whereas another component providing the same service is not. If now the less forgiving one substitutes the original one, the system may not operate anymore. This would affect some of the basic ideas behind component-based development, viz., that of reusability and substitutability of components. A second disadvantage is that this test strategy leads to a strict order in testing of the components, and to a long critical test path. Higher level components cannot be tested before all lower level components have been finished and tested.

Fig. 3(c) shows an alternative test strategy where a lower level component is replaced by a stub or a simulator. Such a stub simulates the basic behavior of the lower level component, providing some functionality of e , typically with hard-coded responses for all requests which i might make on e . The advantage is that components need not to be tested in a strict bottom-up order, but still stubs are typically not powerful enough to guarantee thorough testing of the lower interface behavior of a component, in particular concerning testing of abnormal behavior or robustness. Moreover, stubs have to be developed separately.

The most desirable situation for testing components is depicted in Fig. 3(d): a test environment as a wrapper, or "horse-shoe", around the component with the possibility to fully control and observe all the interfaces of the component. This requires the development of such an environment, and, moreover, the availability of behavior specifications for all these interfaces. The aim of this paper is to work towards this way of testing in a formal context with model-based testing.

Model-based testing of components. For model-based testing of a component in a horse-shoe we need, in principle, a complete model of the behavior of the component specified at all its interfaces. But, as explained above, the specification of a component is usually restricted to the behavior at its upper interface. We indeed assume the availability of a specification of the upper interface of the component under test: $s \in \mathcal{LTS}(L_I^\uparrow, L_U^\uparrow)$. Moreover, instead of having a specification of the lower interface itself, we use the specification of the upper interface of the environmental component that is invoked at the lower interface: $e \in \mathcal{LTS}(L_U^\downarrow, L_I^\downarrow)$. This means that we are not directly testing what the component under test shall do, but what the environmental component expects it to do. Besides, what is missing in these two specifications, and what is consequently also missing in the model-based testing of the component, are the dependencies between the behaviors at the upper and the lower interfaces.

For testing the behavior at the upper interface the **ioco**- or **uioco**-test theory with the corresponding test generation algorithms can directly be used: there is a formal model $s \in \mathcal{LTS}(L_I^\uparrow, L_U^\uparrow)$ from which test cases can be generated, and the implementation is assumed to behave as an input-enabled input-output transition system; see Sect. 2. Moreover, the implementation relations **ioco** and **uioco** seem to express what is intuitively required from a correct implementation at the upper interface: each possible output of the implementation must be included in the outputs of the specification, and also quiescence is only allowed if the specification allows that: a service requester would be disappointed if (s)he would not get an output result if an output is guaranteed in the specification.

For testing the behavior at the lower interface this testing theory is not directly applicable: there is no specification of the required behavior at the lower interface but only a specification of the environment of this lower interface: $e \in \mathcal{LTS}(L_U^\downarrow, L_I^\downarrow)$. This means that we need an implementation relation and a test generation algorithm for such environmental specifications. An issue for such an implementation relation is the treatment of quiescence. Whereas a service requester expects a response when one is specified, a service provider will usually not care when no request is made when this is possible, i.e., the provider does not care about quiescence, but if a request is received it must be a correct request. In the next section we will formally elaborate these ideas, and define the implementation relation for *environmental conformance* **eco**. Subsequently, Sect. 5 will present a test generation algorithm for **eco** including soundness and exhaustiveness, and then Section 6 will briefly discuss the combined testing at the upper- and lower interfaces thus realizing a next step in the "horse-shoe" approach.

4 Environmental Conformance

In this section the implementation relation for environmental conformance **eco** is presented. Referring to Fig. 3(d) this concerns defining the correctness of the behavior of i at its lower interface with respect to what environment specification e expects. Here, we only consider the lower interface of i that communicates with the upper interface of e (or, more precisely, with an implementation k of specification e). Consequently, we use L_I to denote the inputs of i at its lower interface, which are the outputs of e , and L_U to denote the outputs of i at its lower interface, which correspond to the inputs of e . The implementation i is assumed to be input enabled: $i \in \mathcal{IOTS}(L_I, L_U)$; e is just a labelled transition system with inputs and outputs: $e \in \mathcal{LTS}(L_U, L_I)$.

An implementation i can be considered correct with respect to an environment e if the outputs that i produces can be accepted by e , and, conversely, if the outputs produced by e can be accepted by i . Since i is assumed to be input enabled, the latter requirement is trivially fulfilled in our setting. Considering the discussion in Sect. 3, quiescence of i is not an issue here, and consequently it is not considered as a possible output: if i requests a service from e it should do so in the correct way, but i is not forced to request a service just because e is ready to accept such a request. Conversely, quiescence of e does matter. The implementation i would be worried if the environment would not give a response, i.e., would be quiescent, if this were not specified. This, however, is an issue of the correctness of the environment implementation k with respect to the environment specification e , which is not of concern for **eco**.

For the formalization of **eco** we first have to define the sets of outputs (without quiescence), and inputs of a labelled transition system. Note that the set of outputs after a trace σ , $uit(p \text{ after } \sigma)$, collects all outputs that a system may nondeterministically execute, whereas for an input to be in $in(p \text{ after } \sigma)$ it must be executable in all nondeterministically reachable states (cf. the classical

may- and *must*-sets for transition systems [4]). This is justified by the fact that outputs are initiated by the system itself, whereas inputs are initiated by the system's environment, so that acceptance of an input requires that such an input is accepted in all possible states where a system can nondeterministically be. The thus defined set of inputs is strongly related to the set of *Utraces* (Def. 7 in Sect. 2), a fact that will turn out to be important for proving the correctness of test generation in Sect. 5.

Definition 11. *Let q be (a state of) an LTS, and let Q be a set of states.*

1. $in(q) =_{\text{def}} \{ a \in L_I \mid q \xrightarrow{a} \}$
2. $in(Q) =_{\text{def}} \bigcap \{ in(q) \mid q \in Q \}$
3. $uit(q) =_{\text{def}} \{ x \in L_U \mid q \xrightarrow{x} \}$
4. $uit(Q) =_{\text{def}} \bigcup \{ uit(q) \mid q \in Q \}$

Proposition 1

1. $in(q \text{ after } \sigma) = \{ a \in L_I \mid \text{not } q \text{ after } \sigma \text{ refuses } \{a\} \}$
2. $uit(q \text{ after } \sigma) = out(q \text{ after } \sigma) \setminus \{ \delta \}$
3. $Utraces(p) = \{ \sigma \in Straces(s) \mid \forall \sigma_1, \sigma_2 \in L_\delta^*, a \in L_I : \sigma = \sigma_1 \cdot a \cdot \sigma_2 \text{ implies } a \in in(p \text{ after } \sigma_1) \}$

Using these definitions we define **eco**: it expresses that after any possible *Utrace* (without quiescence) of the environment e the outputs that implementation i may produce shall be specified inputs in all possible states that e may (nondeterministically) reach.

Definition 12. *Let $i \in \mathcal{IOTS}(L_I, L_U)$, $e \in \mathcal{LTS}(L_U, L_I)$.*

$$i \text{ eco } e \iff_{\text{def}} \forall \sigma \in Utraces(e) \cap L^* : uit(i \text{ after } \sigma) \subseteq in(e \text{ after } \sigma)$$

Now we have the desired property that after any common behavior of i and e , or of i and k , their outputs are mutually accepted as inputs. As mentioned above, some of these properties are trivial because our implementations are assumed to be input-enabled (we take the "pessimistic view on the environment", cf. [1]).

Definition 13. $p \in \mathcal{LTS}(L_I, L_U)$ and $q \in \mathcal{LTS}(L_U, L_I)$ are mutually receptive i $\forall \sigma \in L^*, \forall x \in L_U, \forall a \in L_I, \forall p', q'$ we have

$$p \parallel q \xrightarrow{\sigma} p' \parallel q' \text{ implies } \left(\begin{array}{l} p' \xrightarrow{!x} \text{ implies } q' \xrightarrow{?x} \\ \text{and } q' \xrightarrow{!a} \text{ implies } p' \xrightarrow{?a} \end{array} \right)$$

Proposition 2. *Let $i \in \mathcal{IOTS}(L_I, L_U)$, $e \in \mathcal{LTS}(L_U, L_I)$, $k \in \mathcal{IOTS}(L_U, L_I)$.*

1. $i \text{ eco } e$ implies i and e are mutually receptive
2. $i \text{ eco } e$ and $k \text{ uioco } e$ implies i and k are mutually receptive

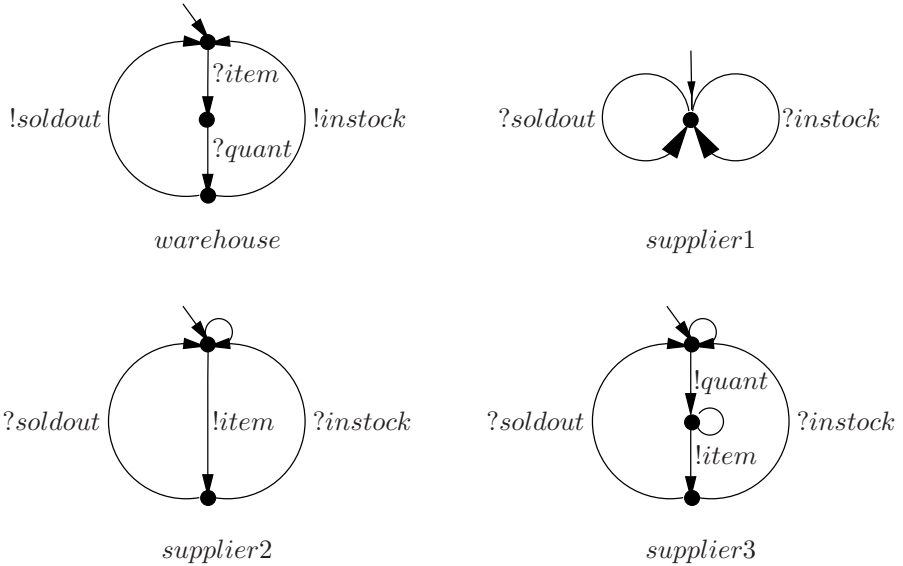


Fig. 4. Exemplifying *eco*

Example 4. To illustrate the **eco** implementation relation, a simple functionality of a *warehouse* component is given in Fig. 4 (top left). For a provided item and quantity the warehouse component reports either $!instock$ or $!soldout$. A supplier component now may use this *warehouse* component to answer requests of customers. Such a supplier implementation must be input enabled for the outputs of the warehouse ($!instock$ and $!soldout$). It communicates via its lower interface with the *warehouse*. The figure shows three supplier implementations; *supplier1* never sends any message to the *warehouse*, it is just input enabled for the possible messages sent from the warehouse. This is fine, we have *supplier1 eco warehouse*, because **eco** does not demand a service requester to really interact with an environmental component. The only demand is that *if* there is communication with the warehouse, then this must be according to the *warehouse* specification.

Bottom left gives *supplier2*. To keep the figures clear, a non-labelled self-loop implicitly represents all input labels that are not explicitly specified, to make a system input-enabled. Here, the service implementer forgot to also inform the *warehouse* of the desired quantity, just the item is passed and then either an $?instock$ or $?soldout$ is expected. What will happen is that *supplier2* will not get any answer from the *warehouse* after having sent the $!item$ message since the *warehouse* waits for the $?quant$ message – both wait in vain. In other words, *supplier2* observes quiescence of the *warehouse*. The *warehouse* does not observe anything since quiescence is not an observation in **eco**. Thus, also here we have *supplier2 eco warehouse*, since this supplier does never send a wrong message to the warehouse. That the *intended transaction* (requesting the

warehouse for an item and quantity, and receiving an answer) is not completed does not matter here; see also Sect. 6. Limitations of **eco**.

Finally, in *supplier3* the implementer confused the order of the messages to be sent to the *warehouse*: instead of sending the *?item* first and then the *?quant* it does it in the reverse order. Here the specification is violated since we have $uit(\text{supplier3 after } \epsilon) = \{!quant\}$ and $in(\text{warehouse after } \epsilon) = \{?item\}$. Hence we get $!quant \notin \{?item\}$, and we have *supplier3 eco warehouse*.

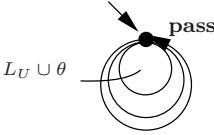
5 Test Generation

Having a notion of correctness with respect to an environmental component, as expressed by the environmental conformance relation **eco**, our next step is to generate test cases for testing implementations according to this relation. Whereas for **ioco** (and **uioco**) test cases are derived from a specification of the implementation under test, test cases for **eco** are not derived from a specification of the implementation but from a specification of the environment of the implementation. The test cases generated from this environment e should check for **eco**-conformance, i.e., they should check after all *Utraces* σ of the environment, whether all outputs produced by the implementation $i - uit(i \text{ after } \sigma) -$ are included in the set of inputs $- in(e \text{ after } \sigma) -$ of e .

Algorithm 1 (eco test generation). Let $e \in \mathcal{LTS}(L_U, L_I)$ be an environmental specification, and let E be a subset of states of e , such that initially $E = e \text{ after } \epsilon$.

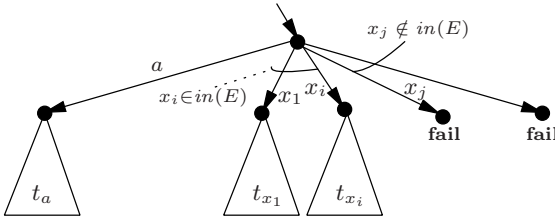
A test case $t \in \mathcal{TTS}(L_U, L_I)$ is obtained from a non-empty set of states E by a finite number of recursive applications of one of the following three nondeterministic choices:

1.



$t := \text{pass}$

2.



$t := a ; t_a$

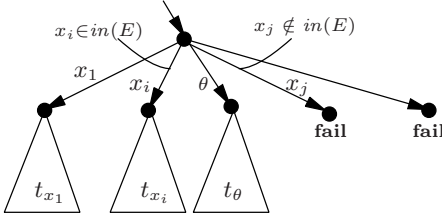
$\square \Sigma \{ x_j ; \text{fail} \mid x_j \in L_U, x_j \notin in(E) \}$

$\square \Sigma \{ x_i ; t_{x_i} \mid x_i \in L_U, x_i \in in(E) \}$

where $a \in L_I$ is an output of e , such that $E \text{ after } a \neq \emptyset$, t_a is obtained by recursively applying the algorithm for the set of states $E \text{ after } a$, and for

each $x_i \in \text{in}(E)$, t_{x_i} is obtained by recursively applying the algorithm for the set of states E **after** x_i .

3.



$$\begin{aligned}
 t &:= \Sigma \{ x_j ; \mathbf{fail} \mid x_j \in L_U, x_j \notin \text{in}(E) \} \\
 &\quad \square \Sigma \{ x_i ; t_{x_i} \mid x_i \in L_U, x_i \in \text{in}(E) \} \\
 &\quad \square \theta ; t_\theta
 \end{aligned}$$

where for each $x_i \in \text{in}(E)$, t_{x_i} is obtained by recursively applying the algorithm for the set of states E **after** x_i , and t_θ is obtained by repeating the algorithm for E .

Algorithm [11](#) generates a test case from a set of states E . This set represents the set of all possible states in which the environment can be at the given stage of the test case generation process. Initially, this is the set e **after** $\epsilon = e_0$ **after** ϵ , where e_0 is the initial state of e . Then the test case is built step by step. In each step there are three ways to make a test case:

1. The first choice is the single-state test case **pass**, which is always a sound test case. It stops the recursion in the algorithm, and thus terminates the test case.
2. In the second choice test case t attempts to supply input a to the implementation, which is an output of the environment. Subsequently, the test case behaves as t_a . Test case t_a is obtained by recursive application of the algorithm for the set E **after** a , which is the set of environment states that can be reached via an a -transition from some current state in E . Moreover, t is prepared to accept, as an input, any output x_i of the implementation, that might occur before a has been supplied. Analogous to t_a , each t_{x_i} is obtained from E **after** x_i , at least if x_i is allowed, i.e., $x_i \in \text{in}(E)$.
3. The third choice consists of checking the output of the implementation. Only outputs that are specified inputs in $\text{in}(E)$ of the environment are allowed; other outputs immediately lead to **fail**. In this case the test case does not attempt to supply an input; it waits until an output arrives, and if no output arrives it observes quiescence, which is always a correct response, since **eco** does not require to test for quiescence.

Now we can state one of our main results: Algorithm [11](#) is sound and exhaustive, i.e., the generated test cases only fail with non-**eco**-conforming implementations, and the test suite consisting of all test cases that can be generated detects all non-**eco**-conforming implementations.

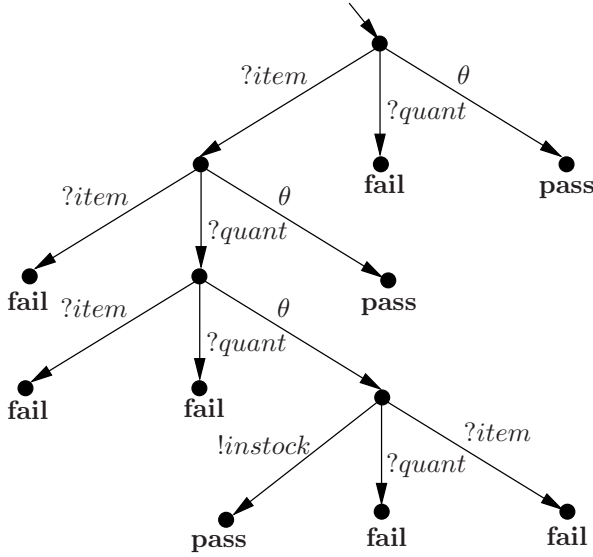


Fig. 5. An **eco** test case derived from the *warehouse* specification

Theorem 2. Let $i \in \mathcal{IOTS}(L_I, L_U)$, $e \in \mathcal{LTS}(L_U, L_I)$, and let $T_e \subseteq \mathcal{TTS}(L_U, L_I)$ be the set of all test cases that can be generated from e with Algorithm 7, then we have

1. Soundness: $i \mathbf{eco} e$ implies $\forall t \in T_e : i \mathbf{passes} t$
2. Exhaustiveness: $i \mathbf{eco} e$ implies $\exists t \in T_e : i \mathbf{fails} t$

Example 5. We continue with Example 4 and Fig. 4, and give an **eco**-test case derived by Algorithm 1 from the *warehouse* specification; see Fig. 5. At the beginning, no input can be applied to the implementation since no outputs are specified in the initial state of the *warehouse*. Note that for the *warehouse* we have inputs $L_U = \{?item, ?quant\}$ and outputs $L_I = \{!instock, !soldout\}$, and that an output from the warehouse is an input to the implementation. First, the third option of the algorithm is chosen (checking the outputs of the implementation). Because $?quant$ is not initially allowed by the *warehouse* this leads to a **fail**. Observing quiescence (θ) is always allowed, and the test case is chosen to stop afterwards via a **pass** (first option). After observing $?item$ the test case continues by again observing the implementation outputs. Because $?item$ is not allowed anymore, since $?item \notin \mathbf{in}(\mathbf{warehouse} \mathbf{after} ?item)$, this leads here to a **fail**. After $?quant$ is observed, again the third option is chosen to observe outputs. Now only quiescence is allowed since there is no implementation output specified in the set $\mathbf{in}(\mathbf{warehouse} \mathbf{after} ?item ?quant)$. Finally, the second option is chosen (applying an input to the implementation). Both $!instock$ and $!soldout$ are possible here. The input $!instock$ to i is chosen, and then the test case is chosen to end with **pass**.

6 Combining Upper and Lower Interface Testing

For testing the behavior at the lower interface of a component implementation i , we proposed the implementation relation **eco** in Sect. 4 with corresponding test generation algorithm in Sect. 5. For testing the behavior at the upper interface we proposed in Sect. 3 to use one of the existing implementation relations **uioco** or **ioco** with one of the corresponding test generation algorithms; see Sect. 2. Since **uioco** is based on *Utraces* like **eco**, whereas **ioco** uses *Straces*, it seems more natural and consistent to choose **uioco** here. Some further differences between **uioco** and **ioco** were discussed in Examples 2 and 3; more can be found in 3.

Now we continue towards testing the whole component implementation in the "horse-shoe" test architecture of Fig. 3(d). This involves concurrently testing for **uioco**-conformance to the provided service specification s at the upper interface, and for **eco**-conformance to the environment specification e at the lower interface. Here, we only indicate some principles and ideas by means of an example, and by discussing the limitations of **eco**. A more systematic treatment is left for further research.

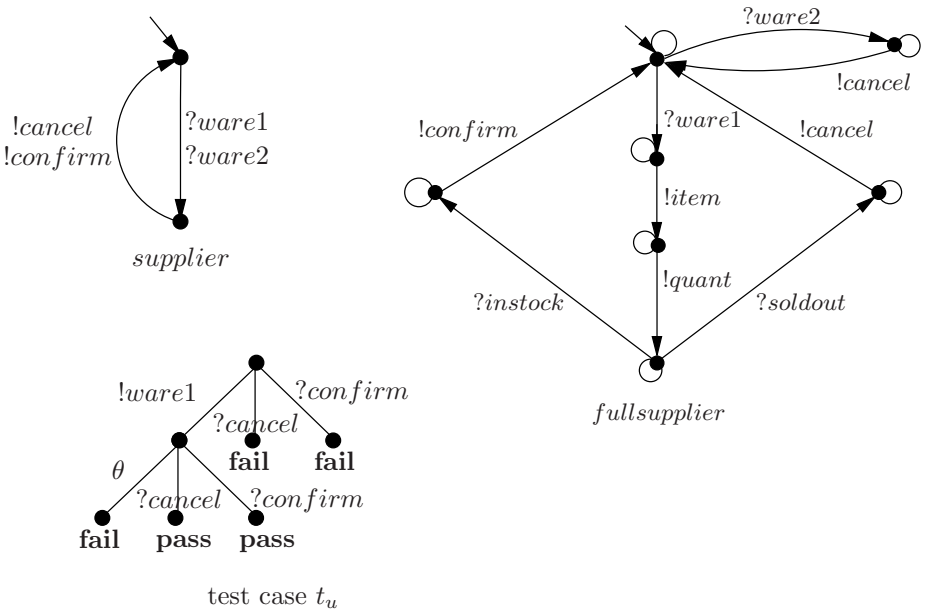


Fig. 6. An upper supplier specification, an implementation covering both interfaces, and an upper-interface test case

The specifications of the upper and lower interfaces are more or less independent, and can be considered as acting in a kind of interleaving manner (it is “a kind of” interleaving because s specifies i directly, and e specifies the environment of i , which implies that it does not make sense to just put $s ||| e$, using the

parallel operator \parallel of Sect. 2). This independence also holds for the derived test cases: we can generate the upper and lower test cases independently from s and e , respectively, after which they can be combined in a kind of interleaving manner. We will show this in Example 6.

Example 6. Fig. 6 shows in *supplier* a specification of the upper interface of a supplier component. This supplier can handle two different warehouses, and requests whether items are in stock. This is done by indicating the favored warehouse via a *?ware1* or *?ware2* message at the upper interface. The supplier is supposed to query the indicated warehouse and either return *!confirm* if the item is in stock, or *!cancel* otherwise. We abstract from modeling the specific items since this does not add here.

A supplier implementation called *fullsupplier* is given at the right-hand side of the figure. This supplier is connected at its lower interface with a warehouse component as being specified in Fig. 4. Its label sets are: $L_I^\uparrow = \{?ware1, ?ware2\}$, $L_U^\uparrow = \{!confirm, !cancel\}$, $L_I^\downarrow = \{?instock, ?soldout\}$, $L_U^\downarrow = \{!item, !quant\}$.

Here we deal with both the upper and the lower interface, therefore the *fullsupplier* must be input enabled for both input sets L_I^\uparrow and L_I^\downarrow . For some reason this supplier cannot deal with a second warehouse, that is why it always reports *!cancel* when being invoked via *?ware2*. For *?ware1* it contacts the *warehouse* component, and behaves as assumed.

Fig. 6 also shows a **uioco**-test case t_u for the upper interface; Fig. 5 specified an **eco**-test case for the lower interface. Now we can test the *fullsupplier* in the horse-shoe test architecture by executing both test cases concurrently, in an interleaved manner. Fig. 7 shows the initial part of such a test case. After *!ware1.?item.?quant* it can be continued with lower-interface input *!instock* after which either *?confirm* or *?cancel* shall be observed by the test case. We deliberately did not complete this test case as a formal structure in Fig. 7, since there are still a couple of open questions, in particular, how to combine quiescence observations in an "interleaved" manner: is there one global quiescence for both interfaces, or does each interface have its own local quiescence? Analogous questions occur for **mioco**, which is a variant of **ioco** for multiple channels [7].

Limitations of eco. It is important to note that we are talking here only about local conformance at the upper and lower interfaces, and not about complete correctness of the component implementation i . The latter is not possible, simply, because we do not have a complete specification for i . In particular, as was already mentioned in the introduction, the dependencies between actions occurring at the upper and lower interfaces are not included in our partial specifications s and e . And where there is no (formal) specification of required behavior there will also be no test to check that behavior.

For instance, in Example 6 the *fullsupplier* relates the *ware1* input at its upper interface with a query at the warehouse component at its lower interface. This relation is invisible to **eco** and **uioco/ioco**. In other words, it is not possible to test requirements like "the supplier must contact a specific warehouse component when, and only when being invoked with message *?ware1*". In Fig. 7,

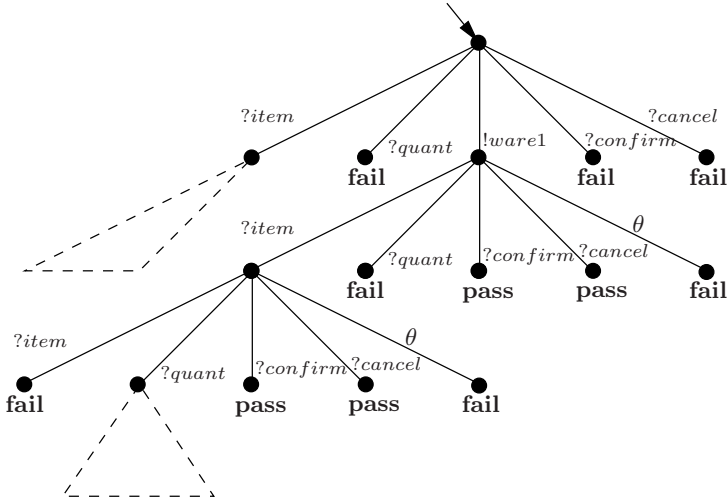


Fig. 7. Initial part of a combined test case

this is reflected in the sequence of actions $!ware1 \cdot ?confirm$, which necessarily leads to **pass**; there is no way to guarantee that the *warehouse* was really queried. In general, requirements like “the supplier queries the right warehouse with the right product” are not testable when only independent specifications of both interfaces are available.

Another noteworthy feature of **eco** is that quiescence cannot be observed by environmental components for several theoretical and practical reasons. For instance, it is not straightforward anymore to indirectly measure quiescence via timeouts here. This again means that a component can always choose to stop communicating with an environmental component. This is not always the desired behavior, since usually a chain of exchanged messages corresponds to a *transaction* that should be entirely performed. For instance, the *warehouse* from Example 4 only gives an answer ($!instock$ or $!soldout$) when being queried with first $?item$ and then $?quant$. Hence, the transaction that the *warehouse* offers, is “send first an item followed by a quantity, and then the availability is returned”. To enforce such transactions, the environmental component must be able to observe quiescence at certain steps within the transaction. For instance, after the reception of $?item$ the requesting service must not be quiescent, it must send $?quant$. Future research might allow to define a transaction-specific notion of quiescence which allows to test also for transactional behavior.

7 Conclusions

When testing a component, standard testing approaches only take the provided interface into account. This is due to the fact that usually only a specification of that interface is available. How the component interacts with environmental

components at its lower interface is not part of the test interest. By so doing, it is not possible to test if a component obeys the specifications of its environment. This is particularly problematic when this misbehavior at the lower interface does not imply an erroneous behavior at the upper interface.

We have introduced a new conformance relation called **eco** which allows to test the lower interface based on specifications of the provided interfaces of the environment. Together with the sound and exhaustive test generation algorithm, this allows to detect such malpractice.

Another important aspect is that a tester for **eco** can be automatically generated from the provided service specifications. In other words, it is possible to generate fully automatic replacements of components which behave according to their specification. This is very useful when implementations of such components are not yet available, or if for reasons like security or safety, a simulated replacement is preferred. The *audition* framework for testing Web Services [2] is currently instantiated with a test engine which combines symbolic versions of the **ioco** [5] and **eco** techniques to allow for sophisticated testing of Service Oriented Architectures.

Modeling and testing components which interact with their environment is not a trivial extension of the standard testing theories like **ioco** for reactive systems. In this paper we pursued the most simple and straightforward path to gain a testing theory which allows for basic testing of both the upper and the lower interface of a component. Though, there are still open questions on how to fully combine **eco** with, for instance, **uioco** or **ioco** on the level of combined specifications and test generation. This should lead to a notion of correctness at the upper interface which takes the lower interface into account. For instance, a deadlock at the lower interface (waiting for a message from an environmental component which never comes) does propagate to quiescence at the upper interface. Also, enriching the lower interface with the ability to observe quiescence of the environment is conceivable.

Finally, important concepts for components are reusability and substitutability. On a theoretical level these correspond to the notions of (pre-)congruences. It was already shown in [3] that without additional restrictions **ioco** is not a precongruence, yet for component based development it is desirable that such properties do hold. More investigations are necessary in this respect, e.g., inspired by the theory of interface automata [1] were such notions like congruence, replaceability, and refinement are the starting point.

Acknowledgments

The inspiration for this work came from two practical projects where model-based testing of components is investigated. In the first, the TANGRAM project (www.esi.nl/tangram), software components of *wafer scanners* developed by ASML (www.asml.com) are tested. It turned out to be difficult to make complete models of components with sufficient detail to perform model-based testing. In the second, the PLASTIC project (EU FP6 IST STREP grant number 26955;

www.ist-plastic.org), one of the research foci is the further elaboration of the *audition* framework with advanced testing methods for services in ubiquitous networking environments.

Lars Frantzen is supported by the EU Marie Curie Network TAROT (MRTN-CT-2004-505121), and by the Netherlands Organization for Scientific Research (NWO) under project STRESS – Systematic Testing of Realtime Embedded Software Systems.

Jan Tretmans carried out this work as part of the TANGRAM project under the responsibility of the Embedded Systems Institute. TANGRAM is partially supported by the Netherlands Ministry of Economic Affairs under grant TSIT2026.

References

1. de Alfaro, L., Henzinger, T.A.: Interface Automata. SIGSOFT Softw. Eng. Notes 26(5), 109–120 (2001)
2. Bertolino, A., Frantzen, L., Polini, A., Tretmans, J.: Audition of Web Services for Testing Conformance to Open Specified Protocols. In: Reussner, R., Stafford, J.A., Szyperski, C.A. (eds.) *Architecting Systems with Trustworthy Components*. LNCS, vol. 3938, pp. 1–25. Springer, Heidelberg (2006)
3. van der Bijl, M., Rensink, A., Tretmans, J.: Compositional Testing with IOCO. In: Petrenko, A., Ulrich, A. (eds.) *FATES 2003*. LNCS, vol. 2931, pp. 86–100. Springer, Heidelberg (2004)
4. De Nicola, R.: Extensional Equivalences for Transition Systems. *Theoretical Computer Science* 24, 211–237 (1987)
5. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A Symbolic Framework for Model-Based Testing. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds.) *Formal Approaches to Software Testing and Runtime Verification*. LNCS, vol. 4262, pp. 40–54. Springer, Heidelberg (2006)
6. Gross, H.-G.: *Component-Based Software Testing with UML*. Springer, Heidelberg (2004)
7. Heerink, L.: *Ins and Outs in Refusal Testing*. PhD thesis, University of Twente, Enschede, The Netherlands (1998)
8. Jard, C., Jéron, T.: TGV: Theory, Principles and Algorithms. *Software Tools for Technology Transfer* 7(4), 297–315 (2005)
9. Rehman, M.J., Jabeen, F., Bertolino, A., Polini, A.: Testing Software Components for Integration: A Survey of Issues and Techniques. In: *Software Testing Verification and Reliability*, John Wiley & Sons, Chichester (2007)
10. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software—Concepts and Tools* 17(3), 103–120 (1996)
11. Tretmans, J.: *Model Based Testing with Labelled Transition Systems*. Technical Report ICIS-R6037, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands (2006)
12. de Vries, R.G., Tretmans, J.: On-the-Fly Conformance Testing using SPIN. *Software Tools for Technology Transfer* 2(4), 382–393 (2000)

Exhaustive Testing of Exception Handlers with Enforcer

Cyrille Artho¹, Armin Biere², and Shinichi Honiden³

¹ Research Center for Information Security (RCIS),
National Institute of Advanced Industrial Science and Technology (AIST),
Tokyo, Japan

² Johannes Kepler University, Linz, Austria

³ National Institute of Informatics, Tokyo, Japan

Abstract. Testing application behavior in the presence of I/O failures is extremely difficult. The resources used for testing usually work without failure. Failures typically cannot be initiated on the test suite level and are usually not tested sufficiently. Essentially, each interaction of the application with the environment can result in a failure. The Enforcer tool identifies such potential failures and automatically tests all relevant outcomes of such actions. It combines the structure of unit tests with coverage information and fault injection. By taking advantage of a unit test infrastructure, performance can be improved by orders of magnitude compared to previous approaches. This paper introduces the usage of the Enforcer tool.

1 Introduction

Testing is a scalable, economic, and effective way to uncover faults in software [28,29]. Even though it is limited to a finite set of example scenarios, it is very flexible and by far the most widespread quality assurance method today. Testing is often carried out without formal rigor. However, coverage measurement tools provide a quantitative measure of the quality of a test suite [14,29]. Uncovered (and thus untested) code may still contain faults. Tested code is known to have executed successfully under at least one occasion. For this reason, it is desirable to test each block of code at least once.

A severe limitation of testing is non-determinism, given by both the thread schedule and the actions of the environment. The Enforcer tool targets non-determinism given by potential I/O failures of the underlying system [2]. For system calls, there are usually two basic outcomes: success or failure. Typically the successful case is easy to test, while the failure case can be nearly impossible to trigger. For instance, simulating network outage is non-trivial. Enforcer does not try to cause underlying system calls to fail, but instead it *injects* a failure into the program at run-time to create the same behavior that would have resulted from a system failure.

Assume that a mechanism exists to identify a test case that executes a particular call. In that case, testing both outcomes of that call will be very efficient,

only requiring a duplication of a particular test case. Existing ad-hoc approaches include factoring out small blocks of code in order to manually test exception handlers, or adding extra flags to conditionals that could trigger outcomes that are normally not reachable by modeling test data alone. Figure 1 illustrates this. In the first example, any exception handling is performed by a special method, which can be tested separately, but does not have access to local variables used by the caller. In the second example, which has inspired our work, the unit test has to set a special flag which causes the exception handling code to run artificially. Enforcer automates the tasks of fault injection and controlling the fault for the selected test cases.

<pre> try { socket = new ServerSocket(); } catch (IOException e) { handleIOException(); // error handling code } </pre>	<pre> try { if (testShouldFail) { throw new IOException(); } socket = new ServerSocket(); } catch (IOException e) { // error handling code } </pre>
Factoring out exception handling.	Extra conditional for testing.

Fig. 1. Two manual approaches for exception handler coverage

Similar tools exist that inject faults into the program and thus improve coverage of exception handlers [7, 17]. However, previous tools have not taken the structure of unit tests into account and thus required re-running the entire test suite for each uncovered exception. Therefore, for m unit tests and n uncovered exceptions, previous approaches had a run-time of $O(m \cdot n)$. Enforcer repeats only one unit test per uncovered exception, yielding a run-time of $O(m + n)$, improving performance by several orders of magnitude [2]. The tool operates in three stages, which are described in detail in previous work [2]:

1. Code instrumentation, at compile time or at class load time. This includes injecting code for coverage measurement and for execution of the repeated test suite.
2. Execution of unit tests. Coverage information is now gathered.
3. Re-execution of certain tests, using selective fault injection. This has to be taken into account by coverage measurement code, in order to require only a single instrumentation step.

This document is organized as follows: Section 2 gives the necessary background about sources of failures considered here, and possible implementation approaches. Section 3 gives an overview of the implementation of the tool, while Section 4 introduces its usage. Section 5 describes related work. Section 6 concludes, and Section 7 outlines future work.

2 Background

This section introduces the necessary terminology used in the rest of this article, covering exceptions, fault injection, and various technologies used by our approach.

2.1 Exceptions

An *exception* as commonly used in many programming languages [19,26,27,33] indicates an extraordinary condition in the program, such as the unavailability of a resource. Exceptions are used instead of error codes to return information about the reason why a method call failed. Java also supports *errors*, which indicate “serious problems that a reasonable application should not try to catch” [19]. A method call that fails may “throw” an exception by constructing a new instance of `java.lang.Exception` or a subtype thereof, and using a `throw` statement to “return” this exception to the caller. At the call site, the exception will override normal control flow. The caller may install an exception *handler* by using the `try/catch` statement. A `try` block includes a sequence of operations that may fail. Upon failure, remaining instructions of the `try` block are skipped, the current method stack frame is replaced by a stack frame containing only the new exception, and control is transferred to the exception handler, indicated in Java by the corresponding `catch` block. This process will also be referred to as *exception handling*.

The usage and semantics of exceptions covers a wide range of behaviors. In Java, exceptions are used to signal the unavailability of a resource (e.g., when a file is not found or cannot be written), failure of a communication (e.g., when a socket connection is closed), when data does not have the expected format, or simply for programming errors such as accessing an array at an illegal index. Two fundamentally different types of exceptions can be distinguished: *Unchecked* exceptions and *checked* exceptions. Unchecked exceptions are of type `RuntimeException` and do not have to be declared in a method. They typically concern programming errors, such as array bounds overflows, and can be tested through conventional means. On the other hand, checked exceptions have to be declared by a method which may throw them. Failure of external operations results in such checked exceptions [7,16]. This work therefore focuses on checked exceptions. For the remainder of this paper, a *checked method call* refers to a call to a method which declared checked exceptions.

When an exception is thrown at run-time, the current stack frame is cleared, and its content is replaced with a single instance of type `Exception` (or a subtype thereof). This mechanism helps our goal in two ways:

- Detection of potentially failed system calls is reduced to the analysis of exceptions.
- No special context data is needed except for information contained in the method signature and the exception.

2.2 Fault Injection

Fault injection [21] refers to influencing program behavior by simulation of failures in hardware or software. In hardware and software, fault injection is most useful when applied to prototypes or implementations. This allows for accurate evaluation of fault handling mechanisms. Fault injection thus also serves to study and measure the quality of exception handling and dependability mechanisms [21].

On a hardware level, fault injection can simulate various kinds of hardware corruption, such as bit flips or stuck-at faults (where the output of a gate is stuck at a particular value). These types of fault injection are important for producing highly reliable hardware, where redundancy can compensate for physical component failures caused by power surges, radiation, or other influences. Certain types of software fault injection have the same target, e. g. simulation of storage data corruption or other machine-level defects [21]. The specific notion of faults in hardware as just described is slightly different from the more common one used in the context of automatic test pattern generation (ATPG). ATPG aims at finding test patterns that verify a chip against faults that are induced during the physical production process [6].

This article targets higher-level problems, arising from failures of system calls. An unexpected result of a system call is typically caused by incorrect parameters or by an underlying network communication problem. Because such failures are difficult to produce, they are often poorly tested. Testing a failure of a system call may take a large amount of effort (during the test process) to set up. In the case of network communications, shutting down network resources will affect other processes on the same host and is therefore not practical in a real-life situation.

Because of these problems, faults are typically injected into an application by an automated tool, which relieves the tester from the burden of injecting faults and capturing their effect. In modern programming languages, such faults manifest themselves on the application level as exceptions [7,16,17]. They can be simulated by modifying the application or library code.

Code instrumentation consists of modifying existing application code or injecting additional code into an application, in order to augment its behavior. The goal is typically to leave the original behavior unchanged, or to change it in a very limited way. It corresponds to a generic form of aspect-oriented programming [22], which organizes code instrumentation into a finite set of operations. *Program steering* [23] allows overriding normal execution flow. Program steering typically refers to altering program behavior using application-specific properties [23], or as schedule perturbation [32], which covers non-determinism in thread schedules. This technique is very similar to fault injection. Instead of faults, either cross-cutting code or randomized delays are injected.

2.3 Problem Scope

A *unit test* is a procedure to verify individual modules of code. A *test harness* executes unit tests. *Test suites* combine multiple unit tests (also called *test cases*)

into a single set. Execution of a single unit test is defined as *test execution*, running all unit tests as *test suite execution*. In this paper, a *repeated test suite* denotes an automatically generated test suite that will re-execute certain unit tests, which will be referred to as *repeated tests*.

Coverage information describes whether a certain piece of code has been executed or not. In this paper, only coverage of checked method calls is relevant. The goal of our work was to test program behavior at each location where exceptions are handled, for each possible occurrence of an exception. This corresponds to the *all-e-deacts* criterion [31]. Treating each checked method call individually allows distinction between exception handling before and after a resource, or several resources, have been allocated. Figure 2 illustrates the purpose of this coverage criterion: In the given `try/catch` block, two operations may fail. Both the `read` and the `close` operation may throw an `IOException`. The application is likely going to be in a very different state before and after reading from the resource, and again after having closed the stream. Therefore, it is desirable to test three possible scenarios: Successful execution of statements (1) and (2), success of statement (1) but failure of statement (2), and failure of statement (1), whereupon statement (2) is never executed. This corresponds to full branch coverage if the same semantics are encoded using `if` statements.

```

try {
    /* (1) */ input = stream.read();
    /* (2) */ stream.close();
} catch (IOException e) {
    // exception handling
}

```

Fig. 2. Illustration of the all-e-deacts coverage criterion

The first source of potential failures considered by our fault injection tool are input/output (I/O) failures, particularly on networks. The problem is that a test environment is typically set up to test the normal behavior of a program. While it is possible to temporarily disable the required resources by software, such as shell scripts, such actions often affect the entire system running, not just the current application. Furthermore, it is difficult and error-prone to coordinate such system-wide changes with a test harness. The same applies to certain other types of I/O failures, such as running out of disk space, packet loss on a User Datagram Protocol (UDP) connection, or communication timeout. While the presence of key actions such as resource deallocations can be checked statically [12,36], static analysis is imprecise in the presence of complex data structures. Testing can analyze the exact behavior.

The second source of potential failures are external programs. It is always possible that a system call fails due to insufficient resources or for other reasons. Testing such failures when interacting with a program through inter-process communication such as pipes is difficult and results in much test-specific code.

Our tool, Enforcer, is written in Java and geared towards failures which are signaled by Java exceptions. Certain operations, such as hash code collisions, are also very difficult to test, but do not throw an exception if they occur. A hash code collision occurs when two distinct data objects have the same hash code. Exhaustive testing of such a collision could be addressed by similar means, but is not covered by our tool that deals with exceptions only.

Finally, there exist hard-to-test operations that are not available in Java: In C programs, pointer arithmetic can be used. The exact address returned by memory allocation cannot be predicted by the application, causing portability and testing problems for low-level operations such as sorting data by their physical address. Other low-level operations such as floating point calculations may also have different outcomes on different platforms.

3 Implementation

Exceptions can be tested manually. This approach guarantees optimal performance but is not practical when many exceptions have to be tested. Therefore, testing should be automated. Automation is possible for unit test suites having idempotent tests. These are tests that can be executed independently many times, producing the same result each time. For such tests, Enforcer automates testing of exceptional outcomes while still maintaining optimal performance.

3.1 Manual Testing of Exceptions

The use of unit tests for fault injection has been inspired by unit testing in the JNuke project. JNuke contains a Java model checker implemented in C. Exceptions or failed system calls are handled using conventional control flow constructs. JNuke heavily uses unit testing to ensure its quality [3]. Test coverage of exceptional outcomes is achieved by manual instantiations of a given test.

Figure 3 illustrates manual fault injection in JNuke. Function `unpack` takes a 32-bit integer value containing a floating point number encoded according to the IEEE standard, and returns its floating point value (lines 6 – 20). Internal functions convert this value to a floating point value of the desired precision. This precision can be configured prior to compilation. For performance reasons, floating point values with a precision of 32 bits are usually represented with the same number of bits on the target machine. However, the way floating point numbers are represented may differ, even for the same size. Therefore, it is possible that an underflow occurs for small values: a small number is truncated to zero. This is very rare, but occurred on some less common hardware architectures in our tests. When an underflow is detected, a brief message is printed to the screen (lines 12 – 18).

Most processors will never generate an underflow for 32-bit floating point numbers because an identical representation is used internally. This makes it impossible to test the “exception handler” (in lines 15 – 17) that is triggered for an underflow.

```

1 /* convert constant pool representation (IEEE 754) to C float */
  /* endian conversion performed by caller */

  static int force_insufficient_precision = 0;
5
  JFloat unpack (unsigned int bytes) {
    JFloat result;

    result = convert_bytes(bytes);
10
    /* check against underflow */
    if (underflow_detected(result, bytes)
        /* manual fault injection */
        || force_insufficient_precision) {
15      fprintf (stderr, "unpack: Cannot represent float value"
                " due to insufficient precision.\n");
        return 0;
    }
    return result;
20 }

  /* test cases */
  int testSuccess (JNukeTestEnv * env) {
    int res;
25    res = !JFloat_compare (unpack(FLOAT_1_0), 1.0);
    return res;
  }

  int testFailure (JNukeTestEnv * env) {
30    int res;
    /* activate fault injection */
    force_insufficient_precision = 1;

    res = !JFloat_compare (unpack(SMALL_FLOAT), 0.0);
35    /* return value always == 0 due to fault injection */

    /* disable fault injection */
    force_insufficient_precision = 0;
    return res;
40 }

```

Fig. 3. Manual testing of a library call that can generate an exception

In order to avoid a gap in test coverage, fault injection was implemented manually. An extra test executes the “exception handler”. The test code is implemented in lines 22 – 40. The first test case, `testSuccess`, succeeds on any platform. The second test case, `testFailure`, is designed to execute the “exception handler” that is triggered on underflow. On some architectures, this happens if a small value known to cause an underflow is used. However, this is not sufficient to test the function in question on most platforms. Therefore, fault injection is used to simulate an underflow (flag `force_insufficient_precision` on lines 14, 32, and 38). This manual approach works fairly well when used sparsely, but carries many intrinsic problems:

- Fault injection code is added inside the application and test code. This reduces readability, and may even interfere with normal code. Non-interference of fault injection code can only be checked through inspection (or the use of preprocessor directives in C).
- Not necessarily the same test case is used to test the success and failure cases.
- Fault injection has to be done manually: Fault injection has to be enabled (and disabled!) at the right points in the test code.

For the remainder of this paper, we will again use Java to illustrate exception handling. Typical code tested by Enforcer uses I/O operations that can throw an exception in Java. Figure 4, which is an extended version of the example in Fig. 1, shows how two unit tests can execute the successful and the failure scenario of a given operation. The application code of this example (lines 1–15) contains a `try` block because the I/O operation in line 10 might fail. Usually, it is successful, so a normal unit test will never execute the corresponding `catch` block.

In order to address this problem, the test code (lines 17 – 27) contains two test cases: The first test case is a conventional unit test, which executes the method in question. Because it is assumed that the I/O operation will return successfully, another test case is added. This test case covers the failure scenario

```

1  class ApplicationCode {
    static boolean socketShouldFail = false;

    public void operationUsingIO() {
5      ServerSocket socket;
        try {
            if (socketShouldFail) {
                throw new IOException();
            }
10         socket = new ServerSocket();
        } catch (IOException e) {
            // exception handling code
        }
    }
15 }

    class TestApplicationCode extends TestCase {
        public void testOperationUsingIO_success() {
20         operationUsingIO();
        }

        public void testOperationUsingIO_failure() {
            ApplicationCode.socketShouldFail = true;
            operationUsingIO();
25         ApplicationCode.socketShouldFail = false;
        }
    }

```

Fig. 4. Manual testing of a library call that can generate an exception

by setting a flag `socketShouldFail` prior to execution of the application code. The artificially induced failure results in an exception being thrown. Again, care has to be taken to reset that flag after test execution. Since that flag is `static` (global), it would affect all subsequent test cases if it was not reset.

In the example in Fig. 4 manual fault injection results in about nine additional lines of code (when not counting whitespace). This 60 % increase is of course an extreme figure that is much lower in average real-world scenarios. In the JNuke project where similar code was used [3], only about 0.25 % of the total code contributes to such testing of I/O failures or similar operations. This figure is rather low because JNuke does not use a lot of I/O operations, and I/O code is limited to 7 out of about 165 classes. Nonetheless, in one class, the extra code needed for a manual approach already affects the readability of the application code, due to various flags and conditionals being introduced. Three methods of the application code of that class have an overhead of over 30 % of extra test code covering I/O failures. This hampers readability and maintainability of that code, and clearly calls for automation of testing I/O failures.

3.2 Automation

Java-based applications using JUnit [25] for unit testing have been chosen as the target for this study. Java bytecode is easy to understand and well-documented. JUnit is widely used for unit testing. In terms of programming constructs, the target consists of any unthrown exceptions, i.e., checked method calls where a corresponding `catch` statement exists and that `catch` statement was not reached from an exception originating from said method call. Only checked exceptions were considered because other exceptions can be triggered through conventional testing [7,16]. Artificially generated exceptions are initialized with a special string denoting that this exception was triggered by Enforcer.

A key goal of the tool is to avoid re-execution of the entire test suite after coverage measurement. In order to achieve this, the test process executes in three stages:

1. Code instrumentation, at compile time or at class load time.
2. Execution of unit tests. Coverage information is now gathered.
3. Re-execution of certain tests, with selective fault injection.

Coverage information gathered during test execution serves to identify a unit test u that executes a checked method call, which could trigger a particular exception e . For each such exception that has not been triggered during normal unit testing, Enforcer then chooses to re-run one unit test (such as u) with exception e being injected. This systematically covers all exceptions by re-running one unit test of choice for each exception.

This approach assumes that unit tests are independent of each other and idempotent. What this means is that a unit test should not depend on data structures that have been set up by a previous unit test. Instead, standardized set-up methods such as `setUp` (in JUnit) must be used. Furthermore, a unit test

should not alter persistent data structures permanently, i. e., the global system state after a test has been run should be equal to the system state prior to that test. This behavior is specified in the JUnit contract [25] and usually adhered to in practice, although it is not directly enforced by JUnit. JUnit does not check against global state changes and executes all unit tests in a pre-determined order. Therefore, it is possible to write JUnit test suites that violate this requirement, and it is up to the test engineer to specify correct unit tests. Tools such as DbUnit [11] can ensure that external data, such as tables in a data base, also fulfill the requirement of idempotency, allowing for repeated test execution.

As a consequence of treating each checked method call rather than just each unit test individually, a more fine-grained behavior is achieved. Each unit test may execute several checked method calls. Our approach allows for re-executing individual unit tests several times within the repeated test suite, injecting a different exception each time. This achieves better control of application behavior, as the remaining execution path after an exception is thrown likely no longer coincides with the original test execution. Furthermore, it simplifies debugging, since the behavior of the application is generally changed in only one location for each repeated test execution. Unit tests themselves are excluded from coverage measurement and fault injection, as exception handlers within unit tests serve for diagnostics and are not part of the actual application.

The intent behind the creation of the Enforcer tool is to use technologies that can be combined with other approaches, such that the system under test can be tested in a way that is as close to the original test setup as possible, while still allowing for full automation of the process. Instrumentation is performed directly on Java bytecode [37]. Injected code has to be designed carefully for the two dynamic stages to work together. Details about the architecture of the tool are described in previous work [2].

3.3 Comparison to Stub-Based Fault Injection

Faults can also be injected at library level. By using a stub that calls the regular code for the successful case and throws an exception for the failure case, code instrumentation would not be necessary. The only change in the code would consist of the “redirection” of the original method call to the stub method. Code modification could therefore be done at the callee rather than at each call site. This way, only library code would be affected, and the application could be run without modification¹.

A stub-based approach is indeed a much simpler way to achieve fault injection. However, it cannot be used to achieve selective fault injection, which is dependent on coverage information. Coverage of exception handlers (`catch` blocks) concerns method calls and their exception handlers. As there are usually several method calls for a given method, coverage of checked method calls can only be measured

¹ This idealizing proposition assumes that the application itself does not provide any library-like functionality, i. e., the application does not interface directly with any system calls or produce other types of exceptions that cannot be tested conventionally.

at the call site, not inside the callee. The same argument holds for code that performs selective fault injection, using run-time information to activate injected exceptions when necessary.

Therefore, most of the functionality heavily depends on the caller context and concerns the call site. While it is conceivable to push such code into method call (into the library code), this would require code that evaluates the caller context for coverage measurement. Such code requires run-time reflection capabilities and would be more complex than the architecture that is actually implemented [2]. Therefore, it does not make sense to implement unit test coverage inside the library.

For this reason, the application code is modified for coverage measurement. By also implementing fault injection at the caller, modification of the callee (the library) is avoided. Avoiding modification of the library makes it possible to use native and signed (tamper-proof) library code for testing. A stub-base approach cannot modify such code.

Coverage data in Enforcer is context-sensitive. It is not only important which blocks of code (which checked method calls and which exception handlers) are executed, but also by which unit test. This requires more information than a typical coverage measurement tool provides. After all, the test to cover previously uncovered exceptions is chosen based on this coverage information. Such a test, if deterministic, is known to execute the checked method call in question. As coverage information includes a reference to a test case, that test case can be used for fault injection.

3.4 Complexity

The complexity incurred by our approach can be divided into two parts: Coverage measurement, and construction and execution of repeated test suites. Coverage is measured for each checked method call. The code which updates run-time data structures runs in constant time. Therefore, the overhead of coverage measurement is proportional to the number of checked method calls that are executed at run-time. This figure is negligible in practice.

Execution of repeated test suites may incur a larger overhead. For each uncovered exception, a unit test has to be re-executed. However, each uncovered exception incurs at most one repeated test. Nested exceptions may require multiple injected faults for a repeated test, but still only one repeated test per distinct fault is required [2]. The key to a good performance is that only one unit test, which is known to execute the checked method call in question, is repeated. If a test suite contains m unit tests and n uncovered exceptions, then our approach will therefore execute $m + n$ unit tests. This number is usually not much larger than m , which makes the tool scalable to large test suites.

Large projects contain thousands of unit tests; previous approaches [7,16,17] would re-execute them all for each possible failure, repeating m tests n times, for a total number of $m \cdot (n + 1)$ unit test executions². Our tool only re-executes

² This includes the original test run where no faults are injected.

one unit test for each failure. This improves performance by several orders of magnitude and allows Enforcer to scale up to large test suites. Moreover, the situation is even more favorable when comparing repeated tests with an ideal test suite featuring full coverage of exceptions in checked method calls. Automatic repeated execution of test cases does not require significantly more time than such an ideal test suite, because the only minor overhead that could be eliminated lies in the instrumented code. Compared to manual approaches, our approach finds faults without incurring a significant overhead, with the additional capability of covering outcomes that are not directly testable.

3.5 Nested Control Structures

Enforcer treats nested exceptions by repeating the same unit test that covered the initial exception. Using an iterative approach, incomplete coverage inside an exception handler can be improved for exception handlers containing checked method calls inside nested `try/catch` statements [2].

However, an exception handler may contain nested control structures other than `try/catch`, such as an `if/else` statement, or more complex control structures. When encountering an `if/else` statement inside an exception handler, Enforcer works as follows: After the initial test run, a chosen unit test is repeated to cover the exception handler in question. This way, the `if` condition is evaluated to one of the two possible outcomes. Unfortunately, the other outcome cannot be tested by repeating the same test case, as a deterministic test always results in the same outcome of the predicate. Program steering cannot be used to force execution of the other half of the `if/else` block: A bit flip in the `if` condition would evaluate its predicate to a value that is inconsistent with the program state [3].

Therefore, a different unit test would have to be chosen, one where the predicate evaluates to a different value. A priori, it cannot be guaranteed that such a unit test exists in the first place, as this would imply a solution to the reachability problem. A search for a solution would therefore have to be exhaustive. An exhaustive search executes all unit tests that could possibly trigger a given exception, hoping that one of the tests (by chance) covers the alternative outcome. This is undesirable, because it negates the performance advantage of Enforcer, which is based on the premise of only choosing one test per exception. Other possible solutions are:

- Manual specification of which unit test to choose for those special cases.
- Refactoring the exception handler in question so it can be tested manually.
- Finally, possible values for which a given `if` condition evaluates to a different value could be generated by perturbing an existing unit test case. Such a guided randomization would be similar to “concolic testing”, which combines concrete and symbolic techniques to generate test cases based on a previous test run [18,30].

³ Because of this, the elegant approach to nested exceptions is defeated for the purpose of treating conventional control structures.

The problem of efficiently treating complex control structures inside exception handlers is therefore still open.

4 Usage of the Enforcer Tool

The Enforcer tool is fully automated. The functionality of the tool is embedded into the application by code instrumentation. Code instrumentation can occur after compilation or at load time. After execution of the normal unit test suite, exception coverage is shown, and repeated tests are executed as necessary. To facilitate debugging, an additional feature exists to reduce the log file output that an application may generate. As experiments show, the Enforcer tool successfully finds faults in real applications.

4.1 Running the Tool

Fundamentally, the three steps required to run the tool (instrumentation, coverage measurement, repeated test execution) can be broken down into two categories: Static code analysis, which instruments method calls that may throw exceptions, and run-time analysis. Run-time analysis includes coverage measurement and re-execution of certain unit tests.

Code instrumentation is entirely static, and can be performed after compilation of the application source code, or at class load time. Enforcer supports both modes of operation. Static instrumentation takes a set of class files as input and produces a set of instrumented files as output. Class files requiring no changes are not copied to the target directory, because the Java classpath mechanism can be used to load the original version if no new version is present.

Alternatively, Enforcer can be invoked at load-time. In this mode, the new Java instrumentation agent mechanism is used [34]. Load-time instrumentation is very elegant in the sense that it does not entail the creation of temporary files and reduces the entire usage of the enforcer tool to just adding one extra command line option. There is no need to specify a set of input class files because each class file is instrumented automatically when loaded at run-time.

For execution of repeated tests, no special reset mechanism is necessary. In JUnit, each test is self-contained; test data is initialized from scratch each time prior to execution of a test. Therefore re-execution of a test just recreates the original data set. After execution of the original and repeated tests, a report is printed which shows the number of executed methods calls that can throw an exception, and the number of executed catch clauses which were triggered by said method calls. If instrumentation occurs at load time, then the number of instrumented method calls is also shown. The Enforcer output is shown once the JUnit test runner has finished (see Fig. 5).

4.2 Evaluation of Results

Figure 5 shows a typical output of the Enforcer tool. First, the initial test suite is run. If it is deterministic, it produces the exact same output as when run by

```
Time: 0.402
OK (29 tests)
*** Total number of instrumented method calls: 37
*** Total number of executed method calls: 32
*** Total number of executed catch blocks: 20
*** Tests with uncovered catch blocks to execute: 12
.....
Time: 0.301
OK (12 tests)
*** Total number of executed method calls: 32
*** Total number of executed catch blocks: 32
*** Tests with uncovered catch blocks to execute: 0
```

Fig. 5. Enforcer output when running the wrapped JUnit test suite

the normal JUnit test runner. This output, resulting in a dot for each successful test, is not shown in the figure. After the JUnit test suite has concluded, JUnit reports the total run time and the test result (the number of successful and failed tests). After the completed JUnit test run, Enforcer reports exception coverage.

If coverage reported is less than 100 %, a new test suite is created, which improves coverage of exceptions. This repeated test suite is then executed in the same way the original test suite was run, with the same type of coverage measurements reported thereafter.

The output can be interpreted as follows: 37 method calls that declare exceptions were present in the code executed. Out of these, 32 were actually executed, while five belong to untested or dead code. 12 uncovered paths from an executed method call to their corresponding exception handler exist. Therefore, 12 tests are run again; the second run covers the remaining paths. Note that the second run may have covered additional checked method calls in nested `try/catch` blocks. This would have allowed increased coverage by launching another test run to cover nested exceptions [2].

4.3 Suppression of Stack Traces

Exception handlers often handle an exception locally before escalating that exception to its caller. The latter aspect of this practice is sometimes referred to as re-throwing an exception, and is quite common [2]. In software that is still under development, such exception handlers often include some auxiliary output such as a dump of the stack trace. The reasoning is that such handlers are normally not triggered in a production environment, and if triggered, the stack trace will give the developer some immediate feedback about how the exception occurred. Several projects investigated in a previous case study used this development approach [2]. If no fault injection tool is used, then this output will never appear during testing and therefore does not constitute a problem in production usage.

Unfortunately, this methodology also entails that a fault injection tool will generate a lot of output on the screen if it is used on such an application. While

it is desirable to test the behavior of all exception handlers, the output containing the origin of an exception typically does not add any useful information. If an uncaught exception occurs during unit testing, the test in which it occurs is already reported by the JUnit test runner. Conversely, exceptions which are caught and then re-thrown do not have to be reported unless the goal is to get some immediate visual information about when the exception is handled (as debugging output).

The stack trace reported by such debugging output may be rather long, and the presence of many such stack traces can make it difficult to evaluate the new test log when Enforcer is used. Therefore, it may be desirable to suppress all exception stack traces that are directly caused by injected faults. The latest version of the Enforcer tool implements this feature. “Primary” exceptions, which were injected into the code, are not shown; “secondary” exceptions, which result as a consequence of an incorrectly handled injected fault, are reported. In most cases, this allows for a much easier evaluation of the output. Of course it is still possible to turn this suppression feature off in case a complete report is desired.

4.4 Experiments

Table 1 shows the condensed results of experiments performed [2]. For each case, the number of tests, the time to run the tests, and the time to run them under Enforcer (where the outcome of exceptions are tested in addition to normal testing) are shown. Note that out of a certain number of calls to I/O methods, typically only a small fraction are covered by original tests. Enforcer can cover most of the missing calls, at an acceptable run-time overhead of factor 1.5 – 5. (The reason why certain calls are not covered is because tests were not always fully deterministic, due to concurrency problems or unit tests not being idempotent.)

Table 1. Results of unit tests and injected exception coverage

Application or library	# tests	Time [s]	Time, Enforcer [s]	# exec. calls	# unex. catch	Cov. (orig.)	Cov. (Enforcer)	Faults found
Echomine	170	6.3	8.0	61	54	8 %	100 %	9
Informa	119	33.2	166.6	139	136	2 %	80 %	2
jConfig	97	2.3	4.7	169	162	3 %	61 %	1
jZonic-cache	16	0.4	0.7	8	6	25 %	100 %	0
SFUutils	11	76.3	81.6	6	2	67 %	100 %	0
SixBS	30	34.6	94.3	31	28	10 %	94 %	0
Slimdog	10	228.6	n/a ^a	15	14	7 %	n/a	1
STUN	14	0.06	0.7	0	0	0 %	0 %	0
XTC	294	28.8	35.5	112	112	0 %	92 %	0

^a Repeated unit tests could not be carried out successfully because the unit test suite was not idempotent, and not thread-safe. Enforcer found one fault before the test suite had to be aborted due to a deadlock.

With previous tools [7][17], re-execution of the entire test suite would have lead to an overhead proportional to the number of test cases, which is orders of magnitudes higher even for small projects. For instance, the Informa test suite comprises 119 tests taking about 33 seconds to run. When analyzing that test suite with a previous-generation fault injection tool, the entire test suite would have had to be run another 136 times (once for each unexecuted exception), taking at about an hour and a half, rather than three minutes when using Enforcer. Enforcer found 12 faults in the given example applications, using only existing unit tests and no prior knowledge of the applications [2].

5 Related Work

Test cases are typically written as additional program code for the system under test. White-box testing tries to execute as much program code as possible [28]. In traditional software testing, *coverage* metrics such as statement coverage [14][29] have been used to determine the effectiveness of a test suite. The key problem with software testing is that it cannot guarantee execution of parts of the system where the outcome of a decision is non-deterministic. In multi-threading, the thread schedule affects determinism. For external operations, the small possibility of failure makes testing that case extremely difficult. Traditional testing and test case generation methods are ineffective to solve this problem.

5.1 Static Analysis and Model Checking

Static analysis investigates properties “at compile time”, without executing the actual program. Non-deterministic decisions are explored exhaustively by verifying all possible outcomes. An over-approximation of all possible program behaviors is computed based on the abstract semantics of the program [10]. For analyzing whether resources allocated are deallocated correctly, there exist static analysis tools which consider each possible exception location [36].

Model Checking explores the entire behavior of a system by investigating each reachable state. Model checkers treat non-determinism exhaustively. Results of system-level operations have been successfully modeled this way to detect failures in applications [9] and device drivers [5]. Another project whose goal is very close to that of Enforcer verifies proper resource deallocation in Java programs [24]. It uses the Java PathFinder model checker [35] on an abstract version of the program, which only contains operations relevant to dealing with resources.

Static analysis (and model checking, if used on an abstract version of the program) can only cover a part of the program behavior, such as resource handling. For a more detailed analysis of program behavior, code execution (by testing) is often unavoidable. Execution of a concrete program in a model checker is possible, at least in theory [35]. However, model checking suffers from the state space explosion problem: The size of the state space is exponential in the size of the system. In addition to program abstraction, which removes certain low-level or local operations from a program, non-exhaustive model checking can be applied.

By using heuristics during state space exploration, states where one suspects an error to be likely are given preference during the search [20]. The intention is to achieve a high probability of finding faults even if the entire state space is far larger than what can be covered by a model checker.

Such heuristics-based model checking, also called *directed model checking*, shows interesting similarities to our approach. By being able to store a copy of the program state before each non-deterministic decision, it can also selectively test for success or failure of an I/O call. Furthermore, model checking covers all interleavings of thread executions. The difference to fault injection is that model checking achieves such coverage of non-determinism on a more fine-grained level (for individual operations rather than unit tests) and includes non-determinism induced by concurrency. Model checking does not have to re-execute code up to a point of decision, but can simply use a previously stored program state to explore an alternative. However, the overhead of the engine required to store and compare a full program state still makes model checkers much slower than normal execution environments such as a Java virtual machine [4]. Model checkers also lack optimization facilities such as just-in-time compilers, because the work required to implement such optimizations has so far always been beyond the available development capacity.

5.2 Fault Injection

Even though fault injection cannot cover concurrency and has to re-execute a program up to a given state, it scales much better in practice than model checking. Fault injection in software is particularly useful for testing potential failures of library calls [21]. In this context, random fault injection as a black-box technique has shown to be useful on an application level [15]. Because our goal is to achieve a high test coverage, we target white-box testing techniques.

In software, two types of fault injection exist: low-level fault injection and high-level fault injection. Low-level fault injection targets mechanisms on an operating system level, such as timeouts and processor interrupts. Such fault injection tools simulate these mechanisms by leveraging corresponding functionality of the operating system or hardware [8]. Low-level tools are easier to deploy, but inherently more limited than high-level fault injection tools, which focus on the application level, where a failure of underlying system code typically manifests itself as an exception [7,16,17]. High-level tools are more powerful because they can modify the application code itself. In most cases, the higher-level semantics of injected faults makes failure analysis easier.

Java is a popular target for measuring and improving exception handling, as exception handling locations are well defined [19]. Our approach of measuring exception handler coverage corresponds to the *all-e-deacts* criterion [31]. The static analysis used to determine whether checked method calls may generate exceptions have some similarity with a previous implementation of such a coverage metric [17]. However, our implementation does not aim at a precise instrumentation for the coverage metric. We only target checked exceptions, within the method where they occur. As the generated exceptions are created at the caller

site, not in the library method, an interprocedural analysis is not required. Unreachable statements will be reported as instrumented, but uncovered checked method calls. Such uncovered calls never incur an unnecessary test run and are therefore benign, but hint at poor coverage of the test suite. Furthermore, unlike some previous work [17], our tool has a run-time component that registers which unit test may cause an exception. This allows us to re-execute only a particular unit test, which is orders of magnitude more efficient than running the entire test suite for each exception site. Finally, our tool can dynamically discover the need for combined occurrences of failures when exception handling code should be reached. Such a dynamic analysis is comparable to another fault injection approach [7], but the aim of that project is totally different: It analyzes failure dependencies, while our project targets code execution and improves coverage of exception handling code.

Similar code injection techniques are involved in program steering [23], which allows overriding the normal execution flow. However, such steering is usually very problematic because correct execution of certain basic blocks depends on a semantically consistent program state. Thus program steering has so far only been applied using application-specific properties [23], or as schedule perturbation [13][32], which covers non-determinism in thread schedules. Our work is application-independent and targets non-determinism induced by library calls rather than thread scheduling.

6 Conclusions

Calls to system libraries may fail. Such failures are very difficult to test. Our work uses fault injection to achieve coverage of such failures. During initial test execution, coverage information is gathered. This information is used in a repeated test execution to execute previously untested exception handlers. The process is fully automated and still leads to meaningful execution of exception handlers. Unlike previous approaches, we take advantage of the structure of unit tests in order to avoid re-execution an entire application. This makes our approach orders of magnitude faster for large test suites. The Enforcer tool, which implements this approach, has been successfully applied to several complex Java applications. It has executed previously untested exception handlers and uncovered several faults. Furthermore, our approach may even partially replace test case generation.

7 Future Work

Improvements and extensions to the Enforcer tool are still being made. Currently, a potential for false positives exists because the exact type of a method is not always known at instrumentation time. Instrumentation then conservatively assumes that I/O failures are possible in such methods. This precision could be improved by adding a run-time check that verifies the signature of the

actual method called. Of course this would incur some extra overhead. An improvement w.r.t. execution time could be achieved by an analysis of test case execution time, in order to select the fastest test case for re-execution. Furthermore, an analysis of unit test dependencies could help to eliminate selection of problematic unit tests, which violate the JUnit contract specifying that a test should be self-contained and idempotent.

The idea of using program steering to simulate rare outcomes may even be expanded further. Previous work has made initial steps towards verifying the contract required by hashing and comparison functions, which states that equal data must result in equal hash codes, but equal hash codes do not necessarily imply data equality [119]. The latter case is known as a hash code collision, where two objects containing different data have the same hash code. This case cannot be tested effectively since hash keys may vary on different platforms and test cases to provoke such a collision are hard to write for non-trivial hash functions, and practically impossible for hash functions that are cryptographically secure. Other mathematical algorithms have similar properties, and are subject of future work. It is also possible that such ideas can be expanded to coverage of control structures inside exception handlers, which is still an open problem.

Finally, we are very interested in applying our Enforcer tool to high-quality commercial test suites. It can be expected that exception coverage will be incomplete but already quite high, unlike in cases tested so far. This will make evaluation of test results more interesting.

References

1. Artho, C., Biere, A.: Applying static analysis to large-scale, multithreaded Java programs. In: Proc. 13th Australian Software Engineering Conference (ASWEC 2001), Canberra, Australia, pp. 68–75. IEEE Computer Society Press, Los Alamitos (2001)
2. Artho, C., Biere, A., Honiden, S.: Enforcer – efficient failure injection. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, Springer, Heidelberg (2006)
3. Artho, C., Biere, A., Honiden, S., Schuppan, V., Eugster, P., Baur, M., Zweimüller, B., Farkas, P.: Advanced unit testing – how to scale up a unit test framework. In: Proc. Workshop on Automation of Software Test (AST 2006), Shanghai, China (2006)
4. Artho, C., Schuppan, V., Biere, A., Eugster, P., Baur, M., Zweimüller, B.: JNuke: Efficient Dynamic Analysis for Java. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 462–465. Springer, Heidelberg (2004)
5. Ball, T., Podelski, A., Rajamani, S.: Boolean and Cartesian Abstractions for Model Checking C Programs. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 268–285. Springer, Heidelberg (2001)
6. Bushnell, M., Agrawal, V.: Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits. Kluwer Academic Publishers, Dordrecht (2000)
7. Candea, G., Delgado, M., Chen, M., Fox, A.: Automatic failure-path inference: A generic introspection technique for Internet applications. In: Proc. 3rd IEEE Workshop on Internet Applications (WIAPP 2003), Washington, USA, p. 132. IEEE Computer Society Press, Los Alamitos (2003)

8. Carreira, J., Madeira, H., Gabriel Silva, J.: Xception: A technique for the experimental evaluation of dependability in modern computers. *Softw. Engineering* 24(2), 125–136 (1998)
9. Colby, C., Godefroid, P., Jagadeesan, L.: Automatically closing open reactive programs. In: *Proc. SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 1998)*, Montreal, Canada, pp. 345–357 (1998)
10. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Proc. 4th ACM Symposium on Principles of Programming Languages (POPL 1977)*, Los Angeles, USA, pp. 238–252. ACM Press, New York (1977)
11. DBUnit (2007), <http://www.dbunit.org/>
12. Engler, D., Musuvathi, M.: Static analysis versus software model checking for bug finding. In: Steffen, B., Levi, G. (eds.) *VMCAI 2004*. LNCS, vol. 2937, pp. 191–210. Springer, Heidelberg (2004)
13. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: *Proc. 20th IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS 2003)*, Nice, France, p. 286. IEEE Computer Society Press, Los Alamitos (2003)
14. Fenton, N., Pfleeger, S.: *Software metrics: a rigorous and practical approach*, 2nd edn. PWS Publishing Co, Boston, USA (1997)
15. Forrester, J.E., Miller, B.P.: An empirical study of the robustness of windows NT applications using random testing. In: *4th USENIX Windows System Symposium*, Seattle, USA, pp. 59–68 (2000)
16. Fu, C., Martin, R., Nagaraja, K., Nguyen, T., Ryder, B., Wonnacott, D.: Compiler-directed program-fault coverage for highly available Java internet services. In: *Proc. 2003 Int'l Conf. on Dependable Systems and Networks (DSN 2003)*, San Francisco, USA, pp. 595–604 (2003)
17. Fu, C., Ryder, B., Milanova, A., Wonnacott, D.: Testing of Java web services for robustness. In: *Proc. ACM/SIGSOFT Int'l Symposium on Software Testing and Analysis (ISSTA 2004)*, Boston, USA, pp. 23–34 (2004)
18. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: *Proc. ACM Int'l Conf. on Programming Language Design and Implementation (PLDI 2005)*, Chicago, USA, pp. 213–223 (2005)
19. Gosling, J., Joy, B., Steele, G., Bracha, G.: *The Java Language Specification*, 3rd edn. Addison Wesley, Reading (2005)
20. Groce, A., Visser, W.: Heuristics for model checking java programs. *Int'l Journal on Software Tools for Technology Transfer (STTT)* 6(4), 260–276 (2004)
21. Hsueh, M., Tsai, T., Iyer, R.: Fault injection techniques and tools. *IEEE Computer* 30(4), 75–82 (1997)
22. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An overview of AspectJ. In: Knudsen, J.L. (ed.) *ECOOP 2001*. LNCS, vol. 2072, pp. 327–355. Springer, Heidelberg (2001)
23. Kim, M., Lee, I., Sammapun, U., Shin, J., Sokolsky, O.: Monitoring, checking, and steering of real-time systems. In: *Proc. 2nd Int'l Workshop on Run-time Verification (RV 2002)*. ENTCS, vol. 70, Elsevier, Amsterdam (2002)
24. Li, X., Hoover, H., Rudnicki, P.: Towards automatic exception safety verification. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) *FM 2006*. LNCS, vol. 4085, pp. 396–411. Springer, Heidelberg (2006)
25. Link, J., Fröhlich, P.: *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann, San Francisco (2003)
26. Meyer, B.: *Eiffel: the language*. Prentice-Hall, Upper Saddle River (1992)

27. Microsoft Corporation: Microsoft Visual C# .NET Language Reference. Microsoft Press, Redmond, USA (2002)
28. Myers, G.: Art of Software Testing. John Wiley & Sons, Chichester (1979)
29. Peled, D.: Software Reliability Methods. Springer, Heidelberg (2001)
30. Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006)
31. Sinha, S., Harrold, M.: Criteria for testing exception-handling constructs in Java programs. In: Proc. IEEE Int'l Conf. on Software Maintenance (ICSM 1999), Washington, USA, p. 265. IEEE Computer Society Press, Los Alamitos (1999)
32. Stoller, S.: Testing concurrent Java programs using randomized scheduling. In: Proc. 2nd Int'l Workshop on Run-time Verification (RV 2002), Copenhagen, Denmark. ENTCS, vol. 70(4), pp. 143–158. Elsevier, Amsterdam (2002)
33. Stroustrup, B.: The C++ Programming Language, 3rd edn. Addison-Wesley Longman Publishing Co, Boston, USA (1997)
34. Sun Microsystems: Santa Clara, USA. Java 2 Platform (Standard edn.) (J2SE) 1.5 (2004), <http://java.sun.com/j2se/1.5.0/>
35. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering Journal 10(2), 203–232 (2003)
36. Weimer, W., Necula, G.: Finding and preventing run-time error handling mistakes. In: Proc. 19th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2004), Vancouver, Canada, pp. 419–431. ACM Press, New York (2004)
37. White, A.: SERP, an Open Source framework for manipulating Java bytecode, (2002), <http://serp.sourceforge.net/>

Model-Based Test Selection for Infinite-State Reactive Systems*

Bertrand Jeannet¹, Thierry Jéron², and Vlad Rusu²

¹ INRIA Rhône-Alpes, Montbonnot, France
Bertrand.Jeannet@inrialpes.fr

² IRISA/INRIA, Campus de Beaulieu, Rennes, France
{Thierry.Jeron,Vlad.Rusu}@irisa.fr

Abstract. This paper addresses the problem of off-line selection of test cases for testing the conformance of a black-box implementation with respect to a specification, in the context of reactive systems. Efficient solutions to this problem have been proposed in the context of finite-state models, based on the *ioco* conformance testing theory. An extension of these is proposed in the context of infinite-state specifications, modelled as automata extended with variables. One considers the selection of test cases according to test purposes describing abstract scenarios that one wants to test. The selection of program test cases then consists in syntactical transformations of the specification model, using approximate analyses.

1 Introduction and Motivation

Testing is the most used validation technique to assess the correctness of reactive systems. Among the aspects of software that can be tested, e.g., functionality, performance, timing, robustness, etc, the focus is here on conformance testing and specialized to reactive systems. This activity has been precisely described in the context of telecommunication protocols [15].

Conformance testing consists in checking that a black-box implementation of a system, only known by its interface and its interactions with the environment through this interface, behaves correctly with respect to its specification. Conformance testing then relies on experimenting the system with test cases, with the objective of detecting some faults with respect to the specification's external behaviour, or improve the confidence one may have in the implementation.

Despite the importance of the testing activity in terms of time and cost in the software life-cycle, testing practice most often remains ad hoc, costly and of rather poor quality, with severe consequences on the cost and quality of software. One solution to improve the situation is to automatize some parts of the testing activity, using models of software and formal methods. In this context, for more than a decade, *model-based testing* (see e.g., [6]) advocates the use of formal models and methods to formalize this validation activity. The formalization relies on models of specifications, implementations and test cases, a formal

* This paper is partly based on [27][18][28].

definition of conformance, or equivalently a fault model defining non-conformant implementations, test selection algorithms, and properties of generated test cases with respect to conformance.

In the context of reactive systems, the system is specified in a behavioral model which serves both as a basis for test generation, and as an oracle for the assignment of verdicts in test cases. Testing theories based on finite-state models such as automata associated to fault models (see e.g., the survey [20]), or labelled transition systems with conformance relations (see e.g., [29]) are now well understood.

Test generation/selection algorithms have been designed based on these theories, and tools like TorX [2], TGV [16], Gotcha [3] among others, have been developed and successfully used on industrial-size systems.

Despite these advances, some developments are still necessary to improve the automation of test generation. Crucial aspects such as compositionality (see e.g., [30]), distribution, real-time or hybrid behavior have to be taken into account for complex software. In this paper some recent advances made in our research group are reviewed, which cope with models of reactive systems with data.

In this paper, models of reactive systems called Input/Output Symbolic Transition Systems (ioSTS) are considered. These are automata extended with variables, with distinguished input and output actions, and corresponding to reactive programs without recursion. Their semantics can be defined in terms of infinite-state Input/Output Labelled Transition Systems (ioLTS). For ioLTS, the *ioco* testing theory [29] defines conformance as a partial inclusion of external behaviours (suspension traces) of the implementation in those of the specification. Several research works have considered this testing theory and propose test generation algorithms. The focus here is on off-line test selection, where a test case is built from a specification and a test purpose (representing abstract behaviours one wants to test), and further executed on the implementation. Test cases are built directly from the ioSTS model rather than from the enumerated ioLTS semantic model. This construction relies on syntactical transformations of the specification model, guided by an approximation of the set of states co-reachable from a set of final state.

2 Modelling Reactive Systems with Data Using ioSTS

The work presented in this paper targets reactive systems. A model inspired by I/O automata [24] is proposed and called ioSTS for *Input/Output Symbolic Transition Systems*. This model extends labelled transition systems with data, and is adequate as an intermediate model for imperative programs without recursion and communicating with their environment. The syntax is first presented, then its semantics in terms of transition systems, and the section finishes with the definition of the visible behavior of an ioSTS for testing.

2.1 Syntax of the ioSTS Model

An ioSTS is made of variables, input and output actions carrying communication parameters carried by actions, guards and assignments. As will be seen later, this model will serve for specifications, test cases and test purposes. One thus needs a model general enough for all these purposes.

One important feature of this model is the presence of variables, for which some notations need to be fixed. Given a variable v and a set of variables $V = \{v_1, \dots, v_n\}$, \mathcal{D}_v denotes the domain in which v takes its values, and \mathcal{D}_V the product domain $\mathcal{D}_{v_1} \times \dots \times \mathcal{D}_{v_n}$. An element of \mathcal{D}_V is thus a vector of values for the variables in V . The notation \mathcal{D}_v is used for a vector \mathbf{v} of variables. Depending on the context, a predicate $P(V)$ on a set of variables V may be considered either as a set $P \subseteq \mathcal{D}_V$, or as a logical formula, the semantics of which is a function $\mathcal{D}_V \rightarrow \{\text{true}, \text{false}\}$. An assignment for a variable v depending on the set of variables V is a function of type $\mathcal{D}_V \rightarrow \mathcal{D}_v$. An assignment for a set X of variables is then a function of type $\mathcal{D}_V \rightarrow \mathcal{D}_X$.

Definition 1 (ioSTS). *An Input/Output Symbolic Transition System \mathcal{M} is defined by a tuple (V, Θ, Σ, T) where:*

- $V = V_i \cup V_x$ is the set of variables, partitioned into a set V_i of internal variables and a set V_x of external variables.
- Θ is the initial condition. It is a predicate $\Theta \subseteq \mathcal{D}_{V_i}$ defined on internal variables. It is assumed that Θ has a unique solution in \mathcal{D}_{V_i} .
- $\Sigma = \Sigma_? \cup \Sigma_!$ is the finite alphabet of actions. Each action a has a signature $\text{sig}(a)$, which is a tuple of types $\text{sig}(a) = \langle t_1, \dots, t_k \rangle$ specifying the types of the communication parameters carried by the action.
- T is a finite set of symbolic transitions. A symbolic transition $t = (a, \mathbf{p}, G, A)$, also written $[a(\mathbf{p}) : G(\mathbf{v}, \mathbf{p}) ? \mathbf{v}' := A(\mathbf{v}, \mathbf{p})]$, is defined by
 - an action $a \in \Sigma$ and a tuple of (formal) communication parameters $\mathbf{p} = \langle p_1, \dots, p_k \rangle$, which are local to a transition; without loss of generality, it is assumed that each action a always carries the same vector \mathbf{p} , which is supposed to be well-typed w.r.t. the signature $\text{sig}(a) = \langle t_1, \dots, t_k \rangle$; $\mathcal{D}_{\mathbf{p}}$ is denoted by $\mathcal{D}_{\text{sig}(a)}$;
 - a guard $G \subseteq \mathcal{D}_V \times \mathcal{D}_{\text{sig}(a)}$, which is a predicate on the variables (internal and external) and the communication parameters. It is assumed that guards are expressed in a theory in which satisfiability is decidable;
 - an assignment $A : \mathcal{D}_V \times \mathcal{D}_{\text{sig}(a)} \rightarrow \mathcal{D}_{V_i}$, which defines the evolution of the internal variables. $A_v : \mathcal{D}_V \times \mathcal{D}_{\text{sig}(a)} \rightarrow \mathcal{D}_v$ denotes the function in A defining the evolution of the variable $v \in V_i$.

This model is rather standard, except for the distinction between internal and external variables. The external variables allow an ioSTS \mathcal{M}_1 to play the rôle of an observer (used later to formalize test purposes) by inspecting the variables of another ioSTS \mathcal{M}_2 when composed together with it. There is no explicit notion of control location in the model, since the control structure of an automaton can be encoded by a specific program counter variable (this will be the case in all examples).

Example 1. A simple example of ioSTS, that will serve as a running example, is described in Figure 1. The ioSTS \mathcal{S} has two internal variables x and y , plus a program counter pc taking its values in $\{Rx, Ry, Cmp, End\}$. It has one input action in and three output actions end , ok and nok , and a communication parameter p . For readability, inputs are prefixed by ? and outputs by ! in examples and figures, but these marks do not belong to the alphabet.

Initially, x and y are set to 0, and pc to the location Rx . In Rx , the process either sends an output end and stops in End , or waits for an input in , carrying a value of the parameter p which is stored in x , and moves to Ry . In Ry , the value of the input parameter p of in is stored in y and the process moves to Cmp . In Cmp , either $y - x \geq 2$ and the output ok is sent with this difference, or $y - x < 2$ and nok is sent. In both cases the process loops back in Rx .

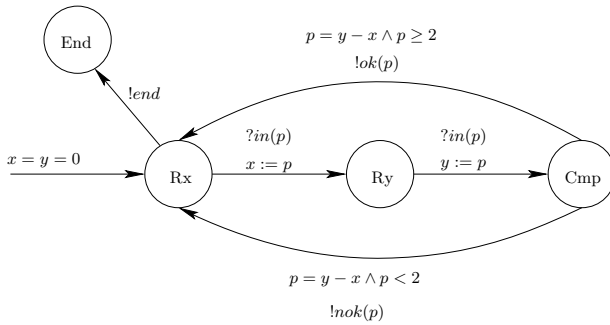


Fig. 1. ioSTS example \mathcal{S}

The use of external variables is motivated by the ioSTS of Figure 4 which represents an observer \mathcal{TP} for \mathcal{S} . It has no internal variable (but could have, e.g. a counter), but has an external variable x observing the internal variable x of \mathcal{S} by synchronization of \mathcal{S} and \mathcal{TP} .

2.2 Semantics of ioSTS

The ioSTS model is a syntactic model allowing to define infinite-state transition systems. The semantics of an ioSTS is an input/output labelled transition systems (ioLTS), i.e. a labelled transition systems (LTS) with distinguished inputs and outputs. This ioLTS semantics is formally defined as follows:

Definition 2 (ioLTS semantics of an ioSTS). *The semantics of an ioSTS $\mathcal{M} = (V, \Theta, \Sigma, T)$ is an ioLTS $\llbracket \mathcal{M} \rrbracket = (Q, Q_0, \Lambda, \rightarrow)$ where:*

- $Q = \mathcal{D}_V$ is the set of states;
- $Q^0 = \{\nu = \langle \nu_i, \nu_x \rangle \mid \nu_i \in \Theta \wedge \nu_x \in \mathcal{D}_{V_x}\}$ is the subset of initial states;
- $\Lambda = \{\langle a, \pi \rangle \mid a \in \Sigma \wedge \pi \in \mathcal{D}_{\text{sig}(a)}\}$ is the set of valued actions partitioned into valued inputs $\Lambda_?$, and valued outputs $\Lambda_!$;

– the transition relation \rightarrow is defined by

$$\frac{(a, \mathbf{p}, G, A) \in T \quad \nu = \langle \nu_i, \nu_x \rangle \in \mathcal{D}_V \quad \pi \in \mathcal{D}_{\text{sig}(a)} \quad \nu' = \langle \nu'_i, \nu'_x \rangle \in \mathcal{D}_V \quad G(\nu, \pi) \quad \nu'_i = A(\nu, \pi)}{\nu \xrightarrow{\langle a, \pi \rangle} \nu'} \quad (\text{Sem})$$

A state of the ioLTS is composed of a valuation of the internal and external variables of the ioSTS. In the initial state, the value of internal variables is uniquely defined by Θ , while the value of external variables is arbitrary. Transitions are labeled by *valued actions* composed of an action name and a valuation of communication parameters. The rule (Sem) says that a transition (a, \mathbf{p}, G, A) of an ioSTS can be fired in a state $\nu = \langle \nu_i, \nu_x \rangle$, if there exists a valuation π of the communication parameters \mathbf{p} such that $\langle \nu, \pi \rangle$ satisfies the guard G ; in such a case, the valued action $\langle a, \pi \rangle$ is taken, the internal variables are assigned new values as specified by the assignment A . External variables behave similarly as *volatile* variables in the C language, by taking arbitrary values when a transition is taken. This reflects the fact that their value is defined by another ioSTS.

The semantics of an ioSTS may be an infinite-state ioLTS, because variables may have infinite domains. These ioLTS may also have infinite branching as communication parameters may also have infinite domains.

Notations, runs and traces. Given this semantics, some notions and properties of ioSTS are defined in terms of their underlying ioLTS semantics. As usual for ioLTS, $q \xrightarrow{\alpha} q'$ is used for $(q, a, q') \in \rightarrow$ and $q \xrightarrow{\alpha}$ for $\exists q', q \xrightarrow{\alpha} q'$. For a sub-alphabet $A' \subseteq A$, a state q of M is said *A'-complete* if $\forall \alpha \in A' : q \xrightarrow{\alpha}$. It is *complete* if it is *A-complete*. The ioLTS $\llbracket \mathcal{M} \rrbracket$ is *A'-complete* (resp. *complete*) if all its states are *A'-complete* (resp. *complete*). Note that these completeness conditions can be defined on ioSTS: an ioSTS \mathcal{M} is Σ' -complete for $\Sigma' \subseteq \Sigma$, if for any $a \in \Sigma'$, $\bigwedge_{(a, \mathbf{p}, G, A) \in T} \neg G$ is unsatisfiable (otherwise said $\forall a \in \Sigma', \bigvee_{(a, \mathbf{p}, G, A) \in T} G = \text{true}$).

Using the ioLTS semantics, one can now define the behavior of an ioSTS. A *run* of an ioSTS \mathcal{M} is an alternate sequence of states and valued actions $\rho = q_0 \alpha_0 q_1 \dots \alpha_{n-1} q_n \in Q^0 \cdot (A \cdot Q)^*$ s.t. $\forall i, q_i \xrightarrow{\alpha_i} q_{i+1}$. For a set $F \subseteq Q$, the run ρ is *accepted in F* if $q_n \in F$. $\text{Runs}(\mathcal{M})$ denotes the set of runs of \mathcal{M} and $\text{Runs}_F(\mathcal{M})$ denotes the set of accepted runs in F . When modelling the testing activity, we consider that variables and locations, thus states of the ioLTS semantics, cannot be observed by the environment. So abstractions of runs have to be considered, where states are abstracted away. A *trace* of a run $\rho \in \text{Runs}(\mathcal{M})$ is the projection $\text{proj}_A(\rho)$ of ρ on actions. $\text{Traces}(\mathcal{M}) \triangleq \text{proj}_A(\text{Runs}(\mathcal{M}))$ denotes the set of traces of \mathcal{M} and $\text{Traces}_F(\mathcal{M}) \triangleq \text{proj}_A(\text{Runs}_F(\mathcal{M}))$ is the set of traces of runs accepted in F .

The notion of trace leads to the notion of determinism and to the determinization operation. Determinism can be defined at the syntactical level for ioSTS.

Definition 3 (Deterministic ioSTS). *An ioSTS $\mathcal{M} = (V, \Theta, \Sigma, T)$ is deterministic if for any action $a \in \Sigma$, and any pair of transitions $t_1 = (a, \mathbf{p}, G_1, A_1)$*

and $t_2 = (a, \mathbf{p}, G_2, A_2)$ carrying the same action, the conjunction of the guards $G_1 \wedge G_2$ is unsatisfiable.

Whether an ioSTS is deterministic or not can thus be decided as soon as satisfiability of guards is decidable. Note that an ioSTS $\mathcal{M} = (V, \Theta, \Sigma, T)$ with only internal variables, i.e., $V = V_i$ (this will be the case for specifications) is deterministic if and only if $\llbracket \mathcal{M} \rrbracket$ is deterministic. When the set of external variables V_x is non-empty this is not true anymore, as assignments do not constrain V_x .

For the sake of simplicity, we already restricted our attention to ioSTS with no internal actions. We focus further to deterministic ioSTS specifications. Internal actions can be handled if there is no loop of internal actions, and non-deterministic ioSTS may be handled, at least for a sub-class of ioSTS where non-determinism can be solved with bounded look-ahead. For this class, there is a determinization procedure that transforms a non-deterministic ioSTS in a deterministic one with same set of traces [19].

2.3 Visible Behaviour for Testing

During conformance testing, the tester stimulates inputs of the system under test, and observes its outputs. In testing practice, absence of output, called *quiescence*, is also observed using timers, with the assumption that timeout values are large enough such that, if a timeout occurs, the system is indeed quiescent. The tester should indeed be able to distinguish between specified and unspecified quiescence. But as trace semantics does not preserve quiescence in general, possible quiescence should be made explicit on the specification. This transforms traces into *suspension* traces [29] (i.e., traces with possible quiescence between actions). For ioLTS, the transformation, denoted Δ consists in adding a self-loop labelled with a new output δ in each quiescent state. Suspension is defined as follows for ioSTS with the expected effect on the ioLTS semantics (i.e. $\llbracket \Delta(\mathcal{M}) \rrbracket = \Delta(\llbracket \mathcal{M} \rrbracket)$):

Definition 4 (Suspension for ioSTS). For an ioSTS $\mathcal{M} = (V, \Theta, \Sigma, T)$ with alphabet $\Sigma = \Sigma_i \cup \Sigma_o$, the suspension of \mathcal{M} is the ioSTS $\Delta(\mathcal{M}) = (V, \Theta, \Sigma^\delta, T_\delta)$ where

- the alphabet is increased by a new output: $\Sigma^\delta = \Sigma_i^\delta \cup \Sigma_o$ with $\Sigma_i^\delta = \Sigma_i \cup \{\delta\}$,
- new loop transitions labelled by δ are added: $T_\delta = T \cup \{\langle \delta, G_\delta, Id_V \rangle\}$ with

$$G_\delta = \neg \left(\bigvee_{(a, \mathbf{p}, G, A) \in T, a \in \Sigma_i} \exists \pi \in \mathcal{D}_{\text{sig}(a)} : G(\nu, \pi) \right)$$

G_δ evaluates to **true** when no value ν of variables and π of communication parameter can be chosen such that an output can be fired. Transition δ can thus be fired when no output can be fired, and loops in the same state [1].

¹ Note that the satisfiability of G_δ is decidable, as it is the negation of the conjunction of guards G , which satisfiability is assumed decidable.

Example 2. The suspension ioSTS of the ioSTS \mathcal{S} of Figure 1 is represented in Figure 2. In this example guards of δ actions are either **true** in locations *End* and *Ry*, or **false** and discarded in other locations.

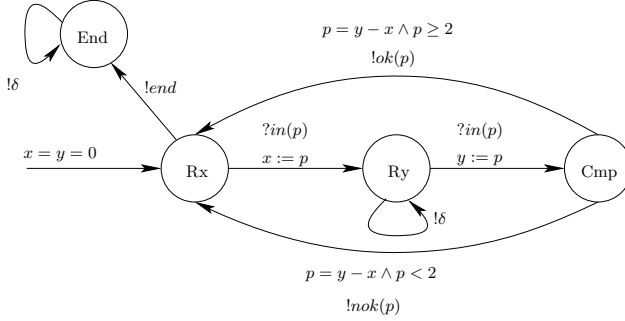


Fig. 2. Suspension ioSTS $\Delta(\mathcal{S})$

For an ioSTS \mathcal{M} modelling a reactive system, the visible behaviour considered for testing is then composed of the traces of its suspension $\Delta(\mathcal{M})$. This is denoted $S\text{Traces}(\mathcal{M}) \triangleq \text{Traces}(\Delta(\mathcal{M}))$. This set of *suspension traces* is considered as the reference behavior for testing conformance with respect to \mathcal{S} .

3 Conformance Testing Theory

The ioco testing theory of [29] can be reformulated in the context of specifications described by ioSTS. This mainly consists in precisising how to model specifications, implementations and test cases, in formally defining conformance as a relation between specification and implementations, and last in modelling test executions and defining their verdicts.

Additionally, properties of test cases that relate verdicts of test executions to conformance should be required: rejection by a test case should mean non-conformance and any non-conformance should be detectable. These properties should be satisfied by the test cases which are automatically generated by our algorithms.

In terms of models for specifications, implementations and test cases, the following is assumed:

- the specification is a deterministic ioSTS $\mathcal{S} = (V^S, \Theta^S, \Sigma, T^S)$, with $\Sigma = \Sigma_! \cup \Sigma_?$ and $V_x^S = \emptyset$ (\mathcal{S} has only internal variables), with ioLTS semantics $\llbracket \mathcal{S} \rrbracket = S = (Q, Q^0, \Lambda, \rightarrow)$ with $\Lambda = \Lambda_! \cup \Lambda_?$.
- the implementation is modelled by a (possibly non-deterministic) ioLTS $I = (Q_I, Q_I^0, \Lambda_I \cup \Lambda_I?, \rightarrow_I)$ having the same interface as \mathcal{S} . I is also assumed to be

$\Lambda_?$ -complete², and let $\Delta(I)$ be its suspension ioLTS. Implementations are indeed unknown, but in order to reason about conformance, specification models need to be related to models of implementations. This is the classical test hypothesis.

- A test case for the specification ioSTS \mathcal{S} is a deterministic ioSTS $\mathcal{TC} = (V^{TC}, \Theta^{TC}, \Sigma^{TC}, T^{TC})$, where $\Sigma_?^{TC} = \Sigma_!$ and $\Sigma_!^{TC} = \Sigma_?$ (actions are mirrored w.r.t. \mathcal{S}), equipped with a variable **Verdict** $\in V^{TC}$ of the enumerated type $\{\text{none}, \text{fail}, \text{pass}, \text{inconc}\}$. Intuitively, **fail** means rejection, **pass** means that some targeted behaviour has been reached (this will be clarified later) and **inconc** means that targeted behaviours cannot be reached anymore. \mathcal{TC} is assumed to be $\Sigma_?^{TC}$ -complete in all states where **Verdict** = **none**. This means that \mathcal{TC} is ready to react to any output of the implementation, except when a verdict is reached and the execution stops. Let $TC = \llbracket \mathcal{TC} \rrbracket = (Q^{TC}, q_0^{TC}, \Lambda^{TC}, \rightarrow_{TC})$ denote the ioLTS semantics of \mathcal{TC} . One denotes by **Fail** = (**Verdict** = **fail**), **Pass** = (**Verdict** = **pass**), and **Inconc** = (**Verdict** = **inconc**) the subsets of Q^{TC} where verdicts are emitted.

Conformance relation. In this setting, a conformance relation defines the set of correct ioLTS implementations I of an ioSTS specification \mathcal{S} . In this paper, the usual ioco relation of Tretmans [29] is considered. This relation defines conformance as a partial inclusion of suspension traces. A definition which is equivalent to the original one can be given as follows.

Definition 5 (Conformance). *Let I be an implementation and \mathcal{S} a specification of I . The ioco conformance relation is defined as:*

$$I \text{ ioco } \mathcal{S} \triangleq STraces(I) \cap NC_STraces(\mathcal{S}) = \emptyset$$

where $NC_STraces(\mathcal{S}) = STraces(\mathcal{S}) \cdot (A_! \cup \{\delta\}) \setminus STraces(\mathcal{S})$.

The set of traces $NC_STraces(\mathcal{S})$ thus exactly characterizes the set of non-conformant behaviours: I is non-conformant as soon as it may exhibit a suspension trace of \mathcal{S} extended with an unspecified output or unexpected quiescence. Interestingly, this formulation of ioco explicits the fact that conformance to a given specification is indeed a safety property of I in the usual meaning: conformance w.r.t. \mathcal{S} is violated if I exhibits a finite trace in $NC_STraces(\mathcal{S})$.

It is possible to characterize $NC_STraces(\mathcal{S})$ by an ioSTS observer accepting exactly this set of traces. This ioSTS called *canonical tester* is built from $\Delta(\mathcal{S})$ as follows:

Definition 6 (Canonical Tester). *Let $\mathcal{S} = (V^{\mathcal{S}}, \Theta^{\mathcal{S}}, \Sigma, T^{\mathcal{S}})$ be a deterministic ioSTS for the specification and $\Delta(\mathcal{S}) = (V^{\mathcal{S}}, \Theta^{\mathcal{S}}, \Sigma^{\delta}, T_{\delta}^{\mathcal{S}})$ its suspension. The canonical tester for \mathcal{S} is the (deterministic) ioSTS $Can(\mathcal{S}) = (V^{Can}, \Theta^{Can}, \Sigma^{Can}, T^{Can})$ such that*

² This ensures that the composition of I with a test case TC never blocks because of non-implemented inputs.

- $V^{Can} = V^S \cup \{\text{Verdict}\}$ where *Verdict* is of the enumerated type $\{\text{none}, \text{fail}\}$
- $\Theta^{Can} = \Theta^S \wedge \text{Verdict} = \text{none}$;
- $\Sigma_?^{Can} = \Sigma_1^\delta$ and $\Sigma_1^{Can} = \Sigma_?$ (the input-output alphabet is mirrored w.r.t. $\Delta(\mathcal{S})$)
- T^{Can} is defined by the rules:

$$\frac{t \in T^S}{t \in T^{Can}} \quad (\text{Keep } T^S)$$

$$\frac{a \in \Sigma_1^\delta = \Sigma_?^{Can} \quad G_a = \bigwedge_{(a,p,G,A) \in T^S} \neg G}{[a(\mathbf{p}) : G_a(\mathbf{v}, \mathbf{p})? \text{Verdict}' := \text{fail}] \in T^{Can}} \quad (\text{Input-completion})$$

It is easy to see that $Can(\mathcal{S})$ is a test case by itself: it is deterministic and $\Sigma_?^{Can}$ -complete in all states where *Verdict* = **none**. Moreover it exactly characterizes non-conformant behaviours as $Traces_{\text{Fail}}(Can(\mathcal{S})) = NC_S Traces(\mathcal{S})$. Consequently conformance can be written:

$$I \text{ ioco } \mathcal{S} \iff STraces(I) \cap Traces_{\text{Fail}}(Can(\mathcal{S})) = \emptyset$$

If I was known, verifying conformance could be reduced to a reachability problem: check whether **Fail** is reachable in the synchronous product of ioLTS $I \times \llbracket Can(\mathcal{S}) \rrbracket$. This may be difficult when $\llbracket Can(\mathcal{S}) \rrbracket$ is infinite.

However, as I is unknown, one can only make experiments with selected test cases, providing inputs and checking that outputs and quiescences of I are specified in \mathcal{S} . This entails that, except in simple cases, conformance cannot be proved by testing, only non-conformance witnesses can be exhibited.

Example 3. The Figure 3 represents an abstract view of the canonical tester of the example specification \mathcal{S} of Figure 1. For example, in location *Cmp*, there should be a transition labelled by the input *?ok* with guard $p \neq y - x \vee p < 2$, a transition labelled by the input *?nok* with guard $p \neq y - x \vee p \geq 2$ and a transition labelled by the input *?end* with guard **true**, all these transitions having the assignment *Verdict* := **fail**. Rather than explicitly describing guards of added transitions, all these transitions are represented by a single transition with label *?otherwise* and target location **Fail** from the meta-location composed of all locations.

Test execution. The execution of a test case on an implementation is now considered. This execution is naturally modelled as a composition of processes. As quiescence of I is observed during testing, the composition is with $\Delta(I)$ which explicits this quiescence. Formally, the execution of a test case \mathcal{TC} on an implementation I is modelled by the *parallel composition* of $TC = \llbracket \mathcal{TC} \rrbracket$ with $\Delta(I)$ with synchronization on common actions.

Definition 7 (Test execution). Let $\Delta(I) = (Q^I, Q_0^I, A_I \cup \{\delta\} \cup A_?, \rightarrow_{\Delta(I)})$ and $TC = (Q^{TC}, q_0^{TC}, A_? \cup A_I \cup \{\delta\}, \rightarrow_{TC})$. The test execution of TC on the implementation I is modelled by the parallel composition of $\Delta(I)$ and TC , which

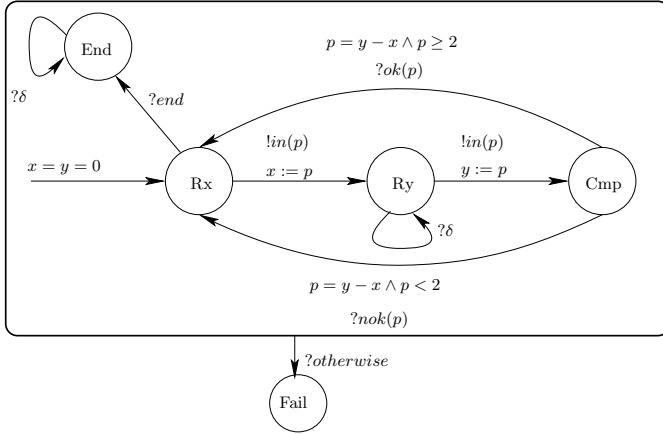


Fig. 3. Canonical tester $Can(\mathcal{S})$

is the ioLTS $\Delta(I) \parallel TC = (Q^I \times Q^{TC}, Q_0^I \times \{q_0^{TC}\}, \Lambda_I \cup \{\delta\} \cup \Lambda_?, \rightarrow_{\Delta(I) \parallel TC})$ where $\rightarrow_{\Delta(I) \parallel TC}$, is defined by the rule:

$$\frac{\alpha \in \Lambda_I \cup \{\delta\} \cup \Lambda_? \quad q_1 \xrightarrow{\alpha}_{\Delta(I)} q_2 \quad q'_1 \xrightarrow{\alpha}_{TC} q'_2}{(q_1, q'_1) \xrightarrow{\alpha}_{\Delta(I) \parallel TC} (q_2, q'_2)}$$

With this definition, it is clear that $Traces(\Delta(I) \parallel TC) = STraces(I) \cap Traces(TC) = STraces(I) \cap Traces(TC)$. For $P \in \{\text{Fail}, \text{Pass}, \text{Inconc}\}$, one also has $Traces_{Q^I \times P}(\Delta(I) \parallel TC) = STraces(I) \cap Traces_P(TC)$.

A test case rejects an implementation when the Verdict variable reaches the value **fail**. The possible rejection of I by TC is then defined by the fact that $\Delta(I) \parallel TC$ may lead to **Fail** in TC : $TC \text{ mayfail } I \triangleq Traces_{Q^I \times \text{Fail}}(\Delta(I) \parallel TC) \neq \emptyset$ which is equivalent to $STraces(I) \cap Traces_{\text{Fail}}(TC) \neq \emptyset$. Similar definitions can be given for *maypass* and *mayinconc*.

Test case properties. Test generation/selection algorithms should produce test cases with properties relating rejection with non-conformance. Formally,

Definition 8 (Soundness, exhaustiveness). Let TS be a set of test cases. TS is said complete if it is both sound and exhaustive where:

- TS is sound $\triangleq \forall I : (I \text{ ioco } S \implies \forall TC \in TS : \neg(TC \text{ mayfail } I))$, i.e., only non-conformant implementations can be rejected by a test case in TS .
- TS is exhaustive $\triangleq \forall I : (\neg(I \text{ ioco } S) \implies \exists TC \in TS : TC \text{ mayfail } I)$, i.e., any non-conformant implementation can be rejected by a test case in TS .

Using the facts that $I \text{ ioco } S$ is equivalent to $STraces(I) \cap Traces_{\text{Fail}}(Can(\mathcal{S})) = \emptyset$ and that $TC \text{ mayfail } I$ is equivalent to $STraces(I) \cap Traces_{\text{Fail}}(TC) \neq \emptyset$, one can prove the following properties:

Proposition 1. *Let TS be a set of test cases for the specification \mathcal{S} , TS is sound iff $\bigcup_{TC \in TS} \text{Traces}_{\text{Fail}}(TC) \subseteq \text{Traces}_{\text{Fail}}(\text{Can}(\mathcal{S}))$, TS is exhaustive iff $\bigcup_{TC \in TS} \text{Traces}_{\text{Fail}}(TC) \supseteq \text{Traces}_{\text{Fail}}(\text{Can}(\mathcal{S}))$.*

The proposition says that sound test cases are sub-observers of $\text{Can}(\mathcal{S})$, and that an exhaustive test suite must reject all implementations rejected by $\text{Can}(\mathcal{S})$, and thus should cover all these non-conformance detections. It immediately follows that the canonical tester $\text{Can}(\mathcal{S})$ alone forms a complete test suite. In some sense $\text{Can}(\mathcal{S})$ is the most general testing process for conformance w.r.t. \mathcal{S} .

Taking a close look at test generation algorithms for *ioCo* which define complete test suites, one notices that all these algorithms produce an infinite number of unfoldings of $\text{Can}(\mathcal{S})$ covering all *Fail* traces.

Despite the nice properties of $\text{Can}(\mathcal{S})$, in practice it cannot be used directly as a test case. In fact, one wants to select individual test cases focused on some particular behaviour. In particular one often avoids the choice between two outputs in a test case, as outputs are controllable by the tester. Selection of a sound test suite will then be based on the selection of sub-behaviours of $\text{Can}(\mathcal{S})$. A consequence of this selection is that exhaustiveness is often lost if only a finite number of test cases is selected. However, the selection algorithm should remain *limit exhaustive*: for any non-conformant implementation, the algorithm should be able to select a test case that could reject this implementation. In other words, the infinite set of test cases that could be selected should be exhaustive. This is important as this guarantees that any non-conformance is detectable. The contrary would mean that the selection algorithm is too weak, or that the conformance relation is too strong compared to the capability of test cases to distinguish between conformant and non-conformant behaviors.

4 Test Selection for ioSTS

In this section, the selection of ioSTS test cases from an ioSTS specification \mathcal{S} is explained. As explained previously, test selection consists in extracting a sub-observer of the non-conformance observer $\text{Can}(\mathcal{S})$. The first step consists in constructing the ioSTS $\text{Can}(\mathcal{S})$, using Definition 6.

4.1 Test Purposes and Test Selection Problem

Several means have been investigated for test selection including random or non-deterministic generation, test purposes, coverage criteria, etc. This paper focuses on the selection of test cases by test purposes. Intuitively, a test purpose describes some abstract behaviours one wants to test. A test purpose is here formally defined as an ioSTS equipped with a set of accepting locations playing the role of a non-intrusive observer.

Definition 9 (Test purpose). *A Test Purpose for a specification ioSTS $\mathcal{S} = (V, \Theta, \Sigma = \Sigma_! \cup \Sigma_?, T)$, is a deterministic ioSTS $\mathcal{TP} = (V^{TP}, \Theta^{TP}, \Sigma^\delta, T^{TP})$ such that*

- $V_x^{TP} = V_i^S$: test purposes are allowed to observe the internal state of \mathcal{S} ;
- $V_i^{TP} \cap V_i^S = \emptyset$ and V_i^{TP} contains a program counter variable pc^{TP} with $\text{accept} \in \mathcal{D}_{pc^{TP}}$. Its set of accepting states is denoted by $\text{Accept} = \{pc^{TP} = \text{accept}\}$.
- \mathcal{TP} should be complete except when $pc^{TP} = \text{accept}$, which means that for any action $a \in \Sigma^\delta$, $pc^{TP} \neq \text{accept} \Rightarrow \bigvee_{(a,p,G,A) \in \mathcal{T}^{TP}} G = \text{true}$. This ensures that \mathcal{TP} does not restrict the runs of \mathcal{S} before they are accepted (if ever).

Example 4. An example of test purpose is described in Figure 4. It specifies that one wants to select behaviors of \mathcal{S} ending with $ok(2)$ when the value of x is greater than 3, without any output nok and no output $ok(p)$ with $p \neq 2$ or when $x < 3$. The label “*” is used for completion, and means “any action with guard being the negation of the conjunction of guards on specified transitions carrying this action”. This test purpose describes behaviors in an abstract way since one can focus on some actions of interest without explicitly specifying intermediate ones.

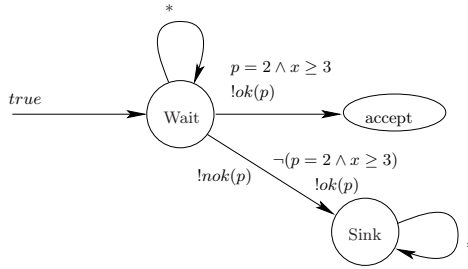


Fig. 4. ioSTS test purpose \mathcal{TP}

The role of a test purpose is to select runs of $\text{Can}(\mathcal{S})$ accepted by \mathcal{TP} . Remember that runs are languages where both actions and states have meanings. The usual way to define such an intersection of languages in the case of ioLTS, is to perform a *synchronous product*. This can be extended to runs by synchronizing states. An operation with similar effect on the ioLTS semantics, can be defined on ioSTS, thus defined at a syntactical level, where transitions with same actions synchronize on the conjunction of their guards, and synchronization of states is preformed by the observation of external variables. A general definition could be given, but it is specialized here to the product of the canonical tester of a specification and a test purpose, for its use in test selection. Formally,

Definition 10 (Synchronous product of ioSTS). Let $\text{Can}(\mathcal{S}) = (V^{Can}, \Theta^{Can}, \Sigma^\delta, T^{Can})$ be the canonical tester of \mathcal{S} and $\mathcal{TP} = (V^{TP}, \Theta^{TP}, \Sigma^\delta, T^{TP})$ a test purpose with $V_x^{TP} = V_i^{Can}$. The synchronous product of $\text{Can}(\mathcal{S})$ and \mathcal{TP} is the ioSTS $\mathcal{P} = \text{Can}(\mathcal{S}) \times \mathcal{TP} = (V^P, \Theta^P, \Sigma^{Can}, T^P)$ where

- $V^P = V_i^P \cup V_x^P$, with $V_i^P = V_i^{Can} \cup V_i^{TP}$ and $V_x^P = \emptyset$;
- $\Theta^P(\langle \mathbf{v}^{Can}, \mathbf{v}^{TP} \rangle) = \Theta^{Can}(\mathbf{v}^{Can}) \wedge \Theta^{TP}(\mathbf{v}^{TP})$;
- T^P is defined by the following inference rule:

$$\frac{\begin{array}{l} [a(\mathbf{p}) : G^c(\mathbf{v}^c, \mathbf{p}) ? (\mathbf{v}_i^c)' := A^c(\mathbf{v}^c, \mathbf{p})] \in T^{Can} \\ [a(\mathbf{p}) : G^t(\mathbf{v}^t, \mathbf{p}) ? (\mathbf{v}_i^t)' := A^t(\mathbf{v}^t, \mathbf{p})] \in T^{TP} \end{array}}{[a(\mathbf{p}) : G^c(\mathbf{v}^c, \mathbf{p}) \wedge G^t(\mathbf{v}^t, \mathbf{p}) ? \\ (\mathbf{v}_i^c)' := A^c(\mathbf{v}^c, \mathbf{p}), (\mathbf{v}_i^t)' := A^t(\mathbf{v}^t, \mathbf{p})] \in T^P}$$

Let \mathcal{P}' be the ioSTS obtained by adding the assignment $\text{Verdict} := \text{pass}$ to all transitions with assignment $pc' := \text{accept}$.

Example 5. The synchronous product of $Can(\mathcal{S})$ and \mathcal{TP} for our running example is (partly) described in Figure 5. Note for example the synchronization on the input ok of guards $p = y - x \wedge p \geq 2$ of \mathcal{S} with $p = 2 \wedge x \geq 3$ from \mathcal{TP} .

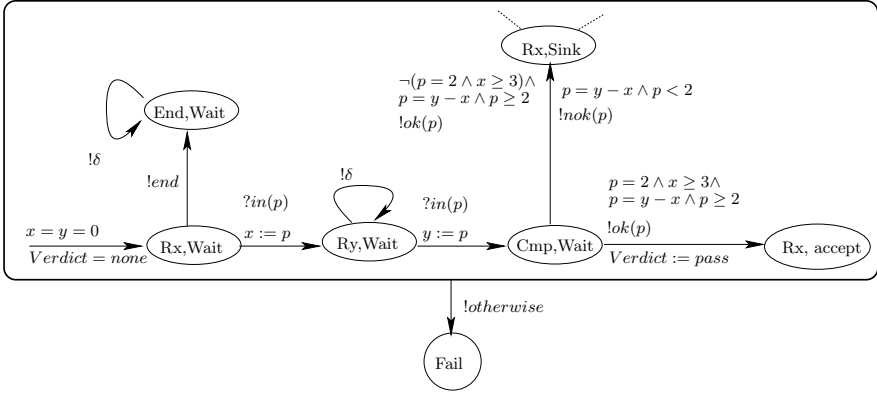


Fig. 5. Synchronous product $Can(\mathcal{S}) \times \mathcal{TP}$

As \mathcal{TP} is non-intrusive (it observes but does not modify \mathcal{S} variables), one gets $Traces(\mathcal{P}') \subseteq Traces(Can(\mathcal{S}))$ and $Traces_{Fail}(\mathcal{P}') = Traces(\mathcal{P}') \cap Traces_{Fail}(Can(\mathcal{S}))$. This means that \mathcal{P}' detects every non-conformance along its traces. It is thus a sound test case.

One also has $Traces_{Pass}(\mathcal{P}') = Traces_{Accept}(\mathcal{P}) \subseteq STraces(\mathcal{S}) \cap Traces_{Accept}(\mathcal{TP})$. This inclusion on traces comes from an equality on runs, lost by projection: even if a run of $Can(\mathcal{S})$ has the same trace as an accepted run, it may be not accepted by \mathcal{TP} because of a condition on its variables observed by \mathcal{TP} . Depending on the considered distinguished states Fail or Pass, the ioSTS observer \mathcal{P}' is both an observer of non-conformant traces and an observer of traces of accepted runs.

It has been shown that \mathcal{P}' is a sound test case. However, as it is an unfolding of $Can(\mathcal{S})$, no selection has been performed yet. Selection is now needed by

focusing on traces accepted in **Pass**. Ideally, one would like to select exactly $Traces_{\text{Pass}}(\mathcal{P}')$, plus unspecified outputs prolonging prefixes of these traces into **Fail** (i.e., traces in $NC_STraces(S)$), denoting detection of non-conformance.

However, the implementations which are considered, even if they are deterministic, are *non-controllable*: their output behaviour is not completely determined by inputs. Thus, after a trace, the tester should consider all possible outputs: those from which **Pass** is reachable or **Fail** is reached, but also those after which **Pass** is not reachable anymore. In this last case, this divergence should be detected as soon as possible, and the **Inconc** verdict should be set.

This reduces to the problem of computing the set $coreach(\text{Pass})$ of states from which **Pass** is reachable. This set can be described by a least fix-point: $coreach(\text{Pass}) = \text{lfp}(\lambda X. \text{Pass} \cup pre(X))$ where $pre(X) = \{q \mid \exists q' \in X, \exists \alpha \in A : q \xrightarrow{\alpha} q'\}$ is the set of states from which X can be reached in one transition. The computation of $coreach(\text{Pass})$ is easy for finite-state systems and can be solved with graph algorithms, as is done in the context of test selection for (finite-state) ioLTS by the TGV tool [16]. However, $coreach(\text{Pass})$ is not computable for *ioSTS* models which have an infinite-state ioLTS semantics. Coping with this computability problem is the subject of the next subsection.

4.2 Approximate Analysis for Test Selection

Faced to this non-computability problem, the proposed solution consists in relying on an over-approximate co-reachability analysis. Using this approximate analysis, the ioSTS \mathcal{P}' is transformed into a test case ioSTS \mathcal{TC} by constraining outputs and detecting inconclusive inputs using syntactical transformations of guards of transitions.

Let $\mathcal{P}' = (V^{P'}, \Theta^{P'}, \Sigma^{Can}, T^{P'})$ as defined by Def. 10. Assume that an over-approximation $coreach^\alpha \supseteq coreach(\text{Pass})$ of the exact set of states co-reachable from **Pass** has been computed. It is assumed that $coreach^\alpha$ is represented by a logical formula.

Moreover, given a set of states $X \in \mathcal{D}_v$ represented by a formula $X(\mathbf{v})$, let $pre(A)(X)(\mathbf{v}, \mathbf{p})$ denote the precondition of X by an assignment $A : \mathcal{D}_v \times \mathcal{D}_p \rightarrow \mathcal{D}_v$:

$$pre(A)(X)(\mathbf{v}, \mathbf{p}) = \exists \mathbf{v}' : X(\mathbf{v}') \wedge \mathbf{v}' = A(\mathbf{v}, \mathbf{p}) = X(A(\mathbf{v}, \mathbf{p}))$$

In other words, $pre(A)(X)(\mathbf{v}, \mathbf{p})$ represents the set of values of variables \mathbf{v} and parameters \mathbf{p} from which X is reached after the assignment A . Note that the operator $pre(A)$ is monotone. Let $pre^\alpha(A)(X) \supseteq pre(A)(X)$ denote a monotone over-approximation of $pre(A)(X)$.

In this context, $pre^\alpha(A)(coreach^\alpha)$ is an over-approximation of the set of values for variables and parameters which allow to stay in $coreach(\text{Pass})$ when taking a transition (a, \mathbf{p}, G, A) , or in other words it is a *necessary condition* to stay in $coreach(\text{Pass})$. Its negation is thus a *sufficient condition* to leave $coreach(\text{Pass})$.

Using these approximate analyses, and the remarks above, a test case can be constructed from \mathcal{P}' as follows. The test case for \mathcal{S} and \mathcal{TP} is the ioSTS

$\mathcal{TC} = (V^{P'}, \Theta^{P'}, \Sigma^{Can}, T^{\mathcal{TC}})$ where $T^{\mathcal{TC}}$ is defined by

$$\frac{(a, \mathbf{p}, G, A) \in T^{P'} \quad a \in \Sigma_!^{Can} \quad G' = pre^\alpha(A)(coreach^\alpha)}{(a, \mathbf{p}, G \wedge G', A) \in T^{\mathcal{TC}}} \quad (\text{Select})$$

$$\frac{(a, \mathbf{p}, G, A) \in T^{P'} \quad a \in \Sigma_?^{Can} \quad A_{\text{Verdict}} = \text{Verdict}' := \text{fail}}{(a, \mathbf{p}, G, A) \in T^{\mathcal{TC}}} \quad (\text{Fail})$$

$$\frac{(a, \mathbf{p}, G, A) \in T^{P'} \quad a \in \Sigma_?^{Can} \quad A_{\text{Verdict}} \neq \text{Verdict}' := \text{fail} \quad G' = pre^\alpha(A)(coreach^\alpha)}{(a, \mathbf{p}, G \wedge G', A), (a, \mathbf{p}, G \wedge \neg G', A') \in T^{\mathcal{TC}}} \quad (\text{Split})$$

where A' is defined by $\begin{cases} A'_{\text{Verdict}} = \text{Verdict}' := \text{inconc}, \\ A'_v = A_v \text{ for } v \neq \text{Verdict}, \end{cases}$

The rule **(Select)** constrains the guards of all *output* transitions such that their post-conditions lead to $coreach^\alpha$, which is an over-approximation of the set of states leading to **pass**. The intuition is that the tester controls its output transitions, thus may restrict them in $G' = pre^\alpha(A)(coreach^\alpha)$ so as to have a chance to stay in the over-approximate co-reachable state-space $coreach^\alpha$ after the transition is fired. The rule **(Fail)** keeps *input* transitions leading to the **fail** verdict. The rule **(Split)** is illustrated by Figure 6. The rule splits the *input* transitions not leading to Fail using a conjunction with guards G' and $\neg G'$: for values of variables and parameters that certainly do not lead to $coreach^\alpha$ (i.e., when $\neg G'$ is true), the **inconc** verdict is emitted, while for values of variables and parameters that may lead to $coreach^\alpha$ (i.e., when G' is true) nothing is changed. The intuition is that input transitions cannot be controlled, but the bad situations from which the verdict **pass** is not reachable may still be detected. In such a case, the verdict **inconc** is emitted. Note that the ioLTS semantics of \mathcal{TC} is different from the semantics of \mathcal{P}' , in particular some output transitions have been removed by rule **(Select)**.

The test case can be further simplified (but without modifying its semantics) with an over-approximation $reach^\alpha(\Theta^{P'})$ of its reachable states $reach(\Theta^{P'})$ where $reach(\Theta^{P'}) = \text{lfp}(\lambda X. \Theta^{P'} \cup post(X))$ with $post(X) = \{q' \mid \exists q \in X, \exists \alpha \in \Lambda : q \xrightarrow{\alpha} q'\}$ being the set of states reachable from X in one transition. The simplification consists in removing transitions of which the guards are unsatisfiable in the over-approximation $reach^\alpha(\Theta^{P'})$ of the set of reachable states i.e., transitions (a, \mathbf{p}, G, A) where $G \wedge reach^\alpha(\Theta^{P'})$ simplifies to **false**.

Example 6. The computation of $coreach^\alpha$ for our running example is described in Figure 7 where the formula is split in boxes attached to each location. The resulting test case, obtained after this co-reachability analysis is represented in Figure 8 (in this figure formulas in boxes will be considered later). In this simple case, an over-approximate analysis based on polyhedra gives an exact result. The effect of the analysis and application of rules is to constrain the guard of the first output *in* with $p \geq 3$ (thus to remove the controllable output *in* with guard $p < 3$ as this certainly leads outside $coreach(\text{Pass})$), and to constrain the

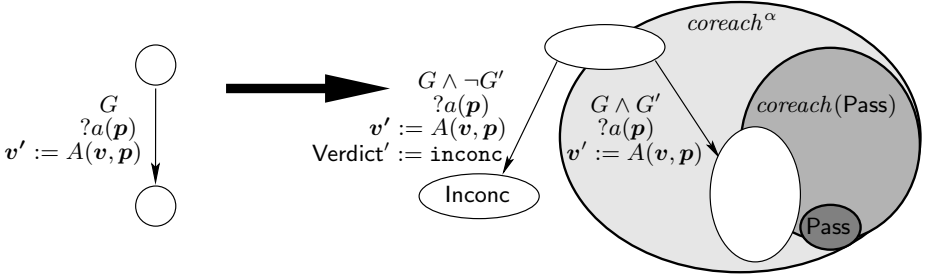


Fig. 6. Illustration of the rule (Split)

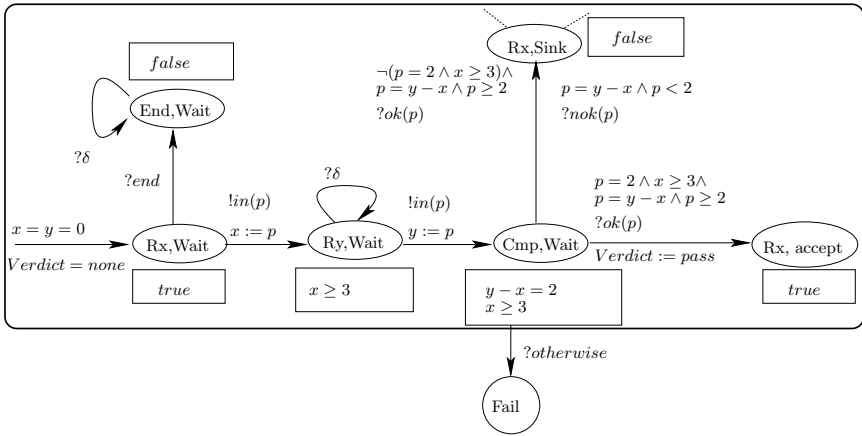


Fig. 7. Computation of $coreach^\alpha$

second output in with $p = x + 2$ (thus remove the controllable output in with $p \neq x + 2$ as this certainly leads outside $coreach(Pass)$).

A reachability analysis on the resulting test case (in boxes attached to locations in Figure 8) allows to further simplify the test case into the one represented in Figure 9). The two transitions from $Cmp, Wait$ to $Inconc$ can be removed: in fact reachability in $Cmp, Wait$ gives $y - x = 2 \wedge x \geq 3$, thus the guard of the transition labelled by nok ($p = y - x \wedge p < 2$) simplifies to **false** in this context, as well as the guard of ok . This illustrates the fact that constraining the guards of outputs using an over-approximate co-reachability analysis can suppress some paths not leading to $Pass$.

In such a case where the analysis is exact, the resulting test case is optimal (but not perfect) with respect to its ability to force the reachability to $Pass$. Nevertheless some uncontrollable actions leading to $Inconc$ may persist: this is the case in our example for the end input which cannot be avoided. A less accurate approximation would give a less selected test case with more possibilities to

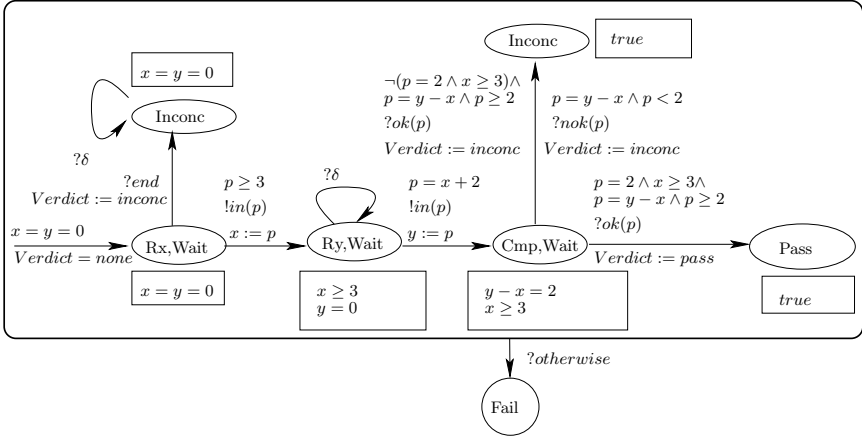


Fig. 8. Resulting test case \mathcal{TC} and computation of $reach(\Theta^{P'})$

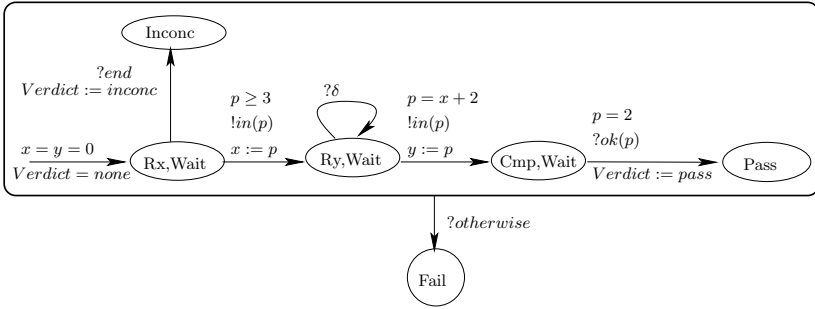


Fig. 9. Resulting test case \mathcal{TC} after simplification

$reach$ $Inconc$ with ok or nok . For example, an analysis which ignores the values of variables would not enforce guards at all, giving as test case the ioSTS P' of Figure 7 with transitions leading to $End, Wait$ and $Rx, Sink$ producing $Inconc$.

4.3 Test Case Properties

It has already been proved that $Can(S)$ is sound. It is also easy to see that this property is preserved by the synchronous product \mathcal{P}' and selection of \mathcal{TC} , as these transformations cannot add any case of rejection. Thus all test cases are sound. Moreover, one can prove the stronger property $Traces_{Fail}(\mathcal{TC}) = Traces(\mathcal{TC}) \cap Traces_{Fail}(Can(S))$, which is preserved from the same property applied to \mathcal{P}' . This property says that only non-conformant implementations can be rejected, and that this rejection happens as soon as possible.

Limit exhaustiveness comes from the following construction: by definition of ioCo , for any non-conformant implementation I , there exists a trace $\sigma.a$ in $S\text{Traces}(I) \cap NC_S\text{Traces}(\mathcal{S})$. The prefix σ is thus a trace of $\Delta(\mathcal{S})$, while $\sigma.a \in \text{Traces}_{\text{Fail}}(\text{Can}(\mathcal{S}))$. As $\Delta(\mathcal{S})$ has no quiescence ($\Delta(\Delta(\mathcal{S})) = \Delta(\mathcal{S})$), there exists an output b such that $\sigma.b$ is in $S\text{Traces}(\mathcal{S})$ and thus in $\text{Traces}(\text{Can}(\mathcal{S}))$ but not in $\text{Traces}_{\text{Fail}}(\text{Can}(\mathcal{S}))$. It then suffices to construct a test purpose \mathcal{TP} such that the trace $\sigma.b$ leads to **Accept**. Now, let \mathcal{TC} be the test case obtained from \mathcal{S} and \mathcal{TP} . One then gets $\sigma.a \in S\text{Traces}(I) \cap \text{Traces}_{\text{Fail}}(\mathcal{TC})$ which by definition means that \mathcal{TC} may reject I .

Soundness and exhaustiveness restrict properties to the relation of **Fail** verdicts to conformance. When using test purposes however, one is also interested in properties of test case verdicts **Pass** and **Inconc** w.r.t. the test purposes. This is where over-approximation has an influence. It is perhaps surprising to see that **Pass** verdicts are always exact in the following sense: $\text{Traces}_{\text{Pass}}(\mathcal{TC}) = \text{Traces}(\mathcal{TC}) \cap \text{Traces}_{\text{Accept}}(\mathcal{P})$. What is lost by the over-approximation of $\text{coreach}(\text{Accept})$, compared with an (hypothetical) exact computation, is the ability to provide the most adequate inputs to the implementation, and the ability to detect infeasible traces to **Accept** as soon as this happens, thus to give **Inconc** verdicts as soon as possible. A detailed study of the influence of the precision of the analysis on the accuracy of test cases is presented in [18]. It is not surprising that the more precise the approximation is, the more accurate test cases are.

4.4 Test Execution

Test cases produced so far are ioSTS. In particular the values of communication parameters of test cases are not instantiated. During test execution, values of communication parameters have to be chosen for outputs of the test cases, among values satisfying the guard (e.g. $p = 5$ for $p \geq 3$ in the example). This is performed by a constraint solver. Conversely, when receiving an input from the implementation, or when observing quiescence, as the test case is input complete and deterministic, one has to check which transition can be fired, by checking the guard with the value of the received communication parameter (e.g. go to **Pass** if $p = 2$, and **Fail** otherwise).

4.5 The STG Tool

The principles of test selection described in this paper are implemented in a new version of the STG tool [8] (see <http://www.irisa.fr/vertecs/software.html#STG>). STG implements the main operations needed for selection: the synchronous product and test selection. This selection is based on approximate co-reachability and reachability analyses. These analyses are provided by an interface with the NBac tool [17] using abstract interpretation [9].

Notice that these analyses can be improved using the dynamic partitioning facility of NBac, allowing to separate locations with respect to the analysis according to some criteria. This has proved very useful in some case studies.

5 Related Work

Recently, attempts have been made to generate test cases from models of reactive systems with data [26,10,11]. The challenge here is to generate test cases without enumerating their state space, but rather by working directly on the higher-level specification models, and thus avoiding the state-space explosion problem.

These models, whether they are called extended finite-state machines or symbolic transition systems are essentially automata that manipulate variables (integers, booleans, aggregate types, ...) and correspond to programs without recursive procedure calls and without memory allocation. Testing theories stay unchanged, being based on the semantics of the models in terms of (infinite) transition systems. But the algorithms must be adapted to cope with data in a symbolic way.

Some pioneering approaches were based on extended finite-state machines (EFSM) (see e.g. [20]), but the data and control parts were mostly treated separately. An exception is the work of [26] in which the authors explore the problem of generating confirming configuration sequences that distinguish a configuration (global state) with a set of configurations, on an EFSM model. They use product of machines, projections on variables and model-checking techniques to derive such sequences.

[10] is an attempt to lift the *ioco* testing theory of LTS to finitely branching Symbolic Transition Systems (STS). In STS, data are specified by algebraic data types. This paper proposes an on-the-fly test generation algorithm à la TorX, based on this specification model. This formalization however does not avoid the (partial) enumeration of the LTS semantics of STS.

Several approaches are based on symbolic execution [5,14] and constraint resolution. In [12], the principle of the DART tool is to combine symbolic execution of a program with random testing. Starting from a random test case, a symbolic execution is computed on the program for this execution, giving rise to a new test case by negating the last condition of the symbolic execution path. By repeating this principle, the main execution paths are covered. The PET tool [13] also uses constraint solving to produce test cases as solutions to path conditions produced from a flow-chart specification and an extended automaton specifying a property. In [11] the authors use symbolic execution on ioSTS specification models to build a symbolic execution tree representing all behaviors of the specification (assuming this is possible), and test purposes or coverage criteria extracted from this execution tree. This is implemented in the Agatha tool [23]. Other approaches, such as [25] or [21], respectively implemented in Gatel and BZ-TT, rely on constraint solving techniques to compute paths to a goal. These approaches are limited to deterministic systems, and consider finite unfoldings of systems by limiting the search depth. In [22] the authors use selection hypotheses combined with operation unfolding for algebraic data types and predicate resolution to produce test cases from Lotos specifications. Compared to our approach, these techniques based on constraint solving may produce more precise test cases. However, constraint solving does not allow to cope with loops as is possible with abstract interpretation, but have to limit the unfolding

to a bounded depth. Nevertheless, these should not be considered as opposite methods, but as complementary ones. Test selection with test purposes using approximate analyses can be seen as a front-end used to select an abstract test case, where information on non-conformance is preserved. Then constraint solving techniques can be used to search for instantiated test cases, by limiting the unfolding of remaining loops.

Some approaches are mostly based on abstraction. In the context of extended LTS, [27] is a pioneering work, an initial attempt of the one presented here. But test selection was based on a very basic abstraction. In [18], a short version of the work presented here was introduced, with emphasis on the relation between test case accuracy and the precision of the approximation. In [28], the approach is combined with verification. Observers of suspension traces are used to describe negation of safety properties, and model-checked on the specification. Even if this does not succeed, test cases are selected from the specification according to these observers, using the approach described here. Selected test cases can then both detect non-conformance and violation of the safety property by the implementation, but also violation of the safety property by the specification if the model-checking phase was not complete.

In most other approaches funded on abstractions, one tries to generate instantiated test cases, i.e., with fixed values for input and output parameters, that exercise particular executions of the system. This is the case for [7], where the idea is first to build an abstraction of a μ CRL specification, to generate abstract test cases by reusing the TGV enumerative technique on this abstraction, and then to concretize these test cases on the concrete specification using constraint solving techniques. In the context of white-box testing, Ball [1] uses a combination of predicate abstraction, reachability analysis and symbolic execution.

6 Conclusion and Perspectives

There is still very little research on model-based test generation which is able to cope with models containing both control and data without enumerating data values.

In the present paper, an approach to the off-line selection of test cases from specification models with control and data (ioSTS) and test purposes specified in the same model has been presented. The main advantage of this test generation technique is to avoid the state explosion problem due to the enumeration of data values. Test selection reduces to syntactical operations on these models and relies on an over-approximate analysis of the co-reachable states to a target location. Test cases are generated in the form of ioSTS, thus representing non-instantiated test programs. During execution of test cases on the implementation, constraint solving is used to choose output data values. For simplicity, the theory exposed in this paper is restricted to deterministic specifications. However, non-deterministic specifications can be taken into account with some restrictions [19].

Among the perspectives of this work, more powerful models of systems with features such as time, recursion and concurrency should be considered. For test

generation, one problem to address in these models is partial observability, which, as for ioSTS, entails the identification of determinizable sub-classes corresponding to applications.

Some ideas of these technique can also be used in other contexts, in particular for structural white box testing where test cases are generated from the source code of the system. One of the main problems of these techniques which is to avoid infeasible paths, could be partly solved by techniques similar to those presented here.

Other challenges are the combination of these techniques with coverage-based test selection. Some attempts have been made to define some test coverage criteria by observers [4], which are very similar to test purposes. A combination with our techniques could be beneficial. Another direction should be to use the dynamic partitioning facility (provided by the tool Nbac used by STG) as an aid for test selection with respect to coverage criteria having a deeper semantic meaning.

Acknowledgment

We wish to thank the Organizing Committee of FMCO'06 for this invitation, the reviewers for their useful remarks, as well as all our colleagues for their participation in the work described in this paper.

References

1. Ball, T.: A theory of predicate-complete test coverage and generation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 2–5. Springer, Heidelberg (2005)
2. Belinfante, A., Feenstra, J., de Vries, R.G., Tretmans, J., Goga, N., Feijs, L., Mauw, S., Heerink, L.: Formal test automation: A simple experiment. In: 12th Int. Workshop on Testing of Communicating Systems, Kluwer Academic Publishers, Dordrecht (1999)
3. Benjamin, M., Geist, D., Hartman, A., Mas, G., Smeets, R., Wolfsthal, Y.: A study in coverage-driven test generation. In: Proceedings of the 36th ACM/IEEE Conference on Design Automation (DAC'99) (1999)
4. Blom, J., Hessel, A., Jonsson, B., Pettersson, P.: Specifying and generating test cases using observer automata. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 137–152. Springer, Heidelberg (2005)
5. Boyer, R.S., Elspas, B., Levitt, K.N.: SELECT: a formal system for testing and debugging programs by symbolic execution. In: Proceedings of the International Conference on Reliable Software, pp. 234–245. ACM Press, New York (1975)
6. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
7. Calamé, J.R., Ioustinova, N., van de Pol, J., Sidorova, N.: Data abstraction and constraint solving for conformance testing. In: Proc. of 12th Asia-Pacific Software Engineering Conference (APSEC'05), Taipei, Taiwan, pp. 541–548 (2005)
8. Clarke, D., Jérón, T., Rusu, V., Zinovieva, E.: STG: a symbolic test generation tool. In: Katoen, J.-P., Stevens, P. (eds.) ETAPS 2002 and TACAS 2002. LNCS, vol. 2280, pp. 470–475. Springer, Heidelberg (2002)

9. Cousot, P., Cousot, R.: Abstract intreprétation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th ACM Symposium on Principles of Programming Languages (POPL'77), Los Angeles, CA, pp. 238–252 (1977)
10. Frantzen, L., Tretmans, J., Willemse, T.: Test generation based on symbolic specifications. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, Springer, Heidelberg (2005)
11. Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic execution techniques for test purpose definition. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, Springer, Heidelberg (2006)
12. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation, pp. 213–223. ACM Press, New York (2005)
13. Gunter, E., Peled, D.: Model checking, testing and verification working together. *Formal Aspects of Computing* 17(2), 201–221 (2005)
14. Howden, W.E.: Theoretical and empirical studies of program testing. In: Proceedings of the 3rd international conference on Software engineering (ICSE '78), pp. 305–311. IEEE Press, Piscataway, NJ (1978)
15. ISO/IEC 9646: Conformance Testing Methodology and Framework (1992)
16. Jard, C., Jérón, T.: TGV: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer (STTT)* (octobre 6, 2004)
17. Jeannet, B.: Dynamic partitioning in linear relation analysis. *Formal Methods in System Design* 23(1), 5–37 (2003)
18. Jeannet, B., Jérón, T., Rusu, V., Zinovieva, E.: Symbolic test selection based on approximate analysis. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, Springer, Heidelberg (2005)
19. Jérón, T., Marchand, H., Rusu, V.: Symbolic determinisation of extended automata. In: 4th IFIP International Conference on Theoretical Computer Science, 2006, Santiago, Chile. SSBM (Springer Science and Business Media) (August 2006)
20. Lee, D., Yannakakis, M.: Principles and Methods of Testing Finite State Machines - A Survey. In: Proceedings of the IEEE, vol. 84(8), IEEE Computer Society Press, Los Alamitos (1996)
21. Legéard, B., Peureux, F., Utting, M.: Automated boundary testing from Z and B. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, Springer, Heidelberg (2002)
22. Lestiennes, G., Gaudel, M.-C.: Testing processes from formal specifications with inputs, outputs and data types. In: 13th International Symposium on Software Reliability Engineering (ISSRE'02), Annapolis, Maryland, IEEE Computer Society Press, Los Alamitos (2002)
23. Lugato, D., Bigot, C., Valot, Y.: Validation and automatic test generation on uml models: the AGATHA approach. *Electronics Notes in Theoretical Computer Science* 66(2) (2002)
24. Lynch, N., Tuttle, M.: Introduction to IO automata. *CWI Quarterly* 3(2) (1999)
25. Marre, B., Arnold, A.: Test sequences generation from LUSTRE descriptions: GATEL. In: 15th IEEE International Conference on Automated Software Engineering (ASE'00), p. 229. IEEE Computer Society, Los Alamitos, CA (2000)
26. Petrenko, A., Boroday, S., Groz, R.: Conforming configurations in EFSM testing. *IEEE Transactions on Software Engineering* 30(1) (2004)

27. Rusu, V., du Bousquet, L., Jéron, T.: An approach to symbolic test generation. In: Grieskamp, W., Santen, T., Stoddart, B. (eds.) IFM 2000. LNCS, vol. 1945, pp. 338–357. Springer, Heidelberg (2000)
28. Rusu, V., Marchand, H., Jéron, T.: Automatic verification and conformance testing for validating safety properties of reactive systems. In: Fitzgerald, J.A., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, Springer, Heidelberg (2005)
29. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools* 17(3), 103–120 (1996)
30. Tretmans, J.: Model-based testing with transition systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4111, Springer, Heidelberg (2006)

Verifying Object-Oriented Programs with KeY: A Tutorial

Wolfgang Ahrendt¹, Bernhard Beckert², Reiner Hähnle¹, Philipp Rümmer¹,
and Peter H. Schmitt³

¹ Department of Computer Science and Engineering,
Chalmers University of Technology and Göteborg University
{ahrendt,reiner,philipp}@chalmers.se

² Department of Computer Science,
University of Koblenz-Landau

beckert@uni-koblenz.de

³ Department of Theoretical Computer Science,
University of Karlsruhe

pschmitt@ira.uka.de

Abstract. This paper is a tutorial on performing formal specification and semi-automatic verification of Java programs with the formal software development tool KeY. This tutorial aims to fill the gap between elementary introductions using toy examples and state-of-art case studies by going through a self-contained, yet non-trivial, example. It is hoped that this contributes to explain the problems encountered in verification of imperative, object-oriented programs to a readership outside the limited community of active researchers.

1 Introduction

The KeY system is the main software product of the KeY project, a joint effort between the University of Karlsruhe, Chalmers University of Technology in Göteborg, and the University of Koblenz. The KeY system is a formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible.

This paper is a tutorial on performing formal specification and semi-automatic verification of Java programs with KeY. There is already a tutorial introduction to the KeY prover that is set at the beginner's level and presupposes no knowledge of specification languages, program logic, or theorem proving [3, Chapt. 10]. At the other end of the spectrum are descriptions of rather advanced case studies [3, Chapt. 14 and 15] that are far from being self-contained. The present tutorial intends to fill the gap between first steps using toy examples and state-of-art case studies by going through a self-contained, yet non-trivial, example. We found few precisely documented and explained, yet realistic, case studies even for other verification systems. Therefore, we believe that this tutorial is of interest in its own right, not only for those who want to know about KeY.

We hope that it can contribute to explain the problems encountered in verification of imperative, object-oriented programs to a readership outside the limited community of active researchers.

We assume that the reader is familiar with the Java programming language, with first-order logic and has some experience in formal specification and verification of software, presumably using different approaches than KeY. Specifications in the Java Modeling Language (JML) [16] and expressions in KeY's program logic Java Card DL [3, Chapt. 3] are explained as far as needed.

In this tutorial we demonstrate in detail how to specify and verify a Java application that uses most object-oriented and imperative features of the Java language. The presentation is such that the reader can trace and understand almost all aspects. To this end, we provided the complete source code and specifications at www.key-project.org/fmco06. We strongly encourage reading this paper next to a computer with a running KeY system. Recently, version 1.0 of the KeY system has been released in connection with the KeY book [3]. The KeY tool is available under GPL and can be freely downloaded from www.key-project.org. Information on how to install the KeY tool can also be found on that web site. The experiments in this tutorial were performed with KeY version 1.1 which has improved support for proof automation. It is available at the location mentioned above together with the case study.

The tutorial is organised as follows: in Section 2 we provide some background on the architecture and technologies employed in the KeY system. In Section 3 we describe the case study that is used throughout the remaining paper. It is impossible to discuss all verification tasks arising from the case study. Therefore, in Section 4, we walk through a typical proof obligation (inserting an element into a datastructure) in detail including the source code, the formal specification of a functional property in JML, and, finally, the verification proof. In Section 5 we repeat this process with a more difficult proof obligation. This time around, we abstract away from most features learned in the previous section in favour of discussing some advanced topics, in particular complex specifications written in Java Card DL, handling of complex loops, and proof modularisation with method contracts. We conclude with a brief discussion.

2 The KeY Approach

The KeY Program Verification System. KeY supports several languages for specifying properties of object-oriented models. Many people working with UML and MDA have familiarity with the specification language OCL (Object Constraint Language), as part of UML 2.0. KeY can also translate OCL expressions to natural language (English and German). Another specification language supported by KeY, which enjoys popularity among Java developers and which we use in this paper, is the Java Modeling Language (JML). Optional plugins of KeY into the popular Eclipse IDE and the Borland Together CASE tool suite are available with the intention to lower initial adoption cost for users with no or little training in formal methods.

The target language for verification in KeY is Java Card 2.2.1. KeY is the only publicly available verification tool that supports the full Java Card standard including the persistent/transient memory model and atomic transactions. Rich specifications of the Java Card API are available both in OCL and JML. Java 1.4 programs that respect the limitations of Java Card (no floats, no concurrency, no dynamic class loading) can be verified as well.

The Eclipse and Together KeY plugins allow to select Java classes or methods that are annotated with formal specifications and both plugins offer to prove a number of correctness judgements such as behavioural subtyping, partial and total correctness, invariant preservation, or frame properties. In addition to the JML/OCL-based interfaces one may supply proof obligations directly on the level of Java Card DL. For this, a stand-alone version of the KeY prover not relying on Eclipse or Together is available.

The program logic Java Card DL is axiomatised in a *sequent calculus*. Those calculus rules that axiomatise program formulas define a symbolic execution engine for Java Card and so directly reflect the operational semantics. The calculus is written in a small domain-specific so-called *taclet* language that was designed for concise description of rules. Taclets specify not merely the logical content of a rule, but also the context and pragmatics of its application. They can be efficiently compiled not only into the rule engine, but also into the automation heuristics and into the GUI. Depending on the configuration, the axiomatisation of Java Card in the KeY prover uses 1000–1300 taclets.

The KeY system is not merely a verification condition generator (VCG), but a theorem prover for program logic that combines a variety of automated reasoning techniques. The KeY prover is distinguished from most other deductive verification systems in that symbolic execution of programs, first-order reasoning, arithmetic simplification, external decision procedures, and symbolic state simplification are interleaved.

At the core of the KeY system is the deductive verification component, which also can be used as a stand-alone prover. It employs a free-variable sequent calculus for first-order Dynamic Logic for Java. The calculus is proof-confluent, i.e., no backtracking is necessary during proof search.

While we constantly strive to increase the degree of automation, user interaction remains inexpedable in deductive program verification. The main design goal of the KeY prover is thus a seamless integration of automated and interactive proving. Efficiency must be measured in terms of user plus prover, not just prover alone. Therefore, a good user interface for presentation of proof states and rule application, a high level of automation, extensibility of the rule base, and a calculus without backtracking are all important features.

Syntax and Semantics of the KeY Logic. The foundation of the KeY logic is a typed first-order predicate logic with subtyping. This foundation is extended with parameterised modal operators $\langle p \rangle$ and $[p]$, where p can be any sequence of legal Java Card statements. The resulting multi-modal program logic is called Java Card Dynamic Logic or, for short, Java Card DL [3, Chapt. 3].

As is typical for Dynamic Logic, Java Card DL integrates programs and formulas within a single language. The modal operators refer to the final state of program p and can be placed in front of any formula. The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds, while $[p] \phi$ does not demand termination and expresses that *if* p terminates, then ϕ holds in the final state. For example, “when started in a state where x is zero, $x++$; terminates in a state where x is one” can be expressed as $x \doteq 0 \rightarrow \langle x++ \rangle (x \doteq 1)$. The states used to interpret formulas are first-order structures sharing a common universe.

The type system of the KeY logic is designed to match the Java type system but can be used for other purposes as well. The logic includes *type casts* (changing the static type of a term) and *type predicates* (checking the dynamic type of a term) in order to reason about inheritance and polymorphism in Java programs [3] Chapt. 2]. The type hierarchy contains the types such as *boolean*, the root reference type *Object*, and the type *Null*, which is a subtype of all reference types. It contains a set of user-defined types, which are usually used to represent the interfaces and classes of a given Java Card program. Finally, it contains several integer types, including both the range-limited types of Java and the infinite integer type \mathbb{Z} .

Besides built-in symbols (such as type-cast functions, equality, and operations on integers), user-defined functions and predicates can be added to the signature. They can be either *rigid* or *non-rigid*. Intuitively, rigid symbols have the same meaning in all program states (e.g., the addition on integers), whereas the meaning of non-rigid symbols may differ from state to state.

Moreover, there is another kind of modal operators called *updates*. They can be seen as a language for describing program transitions. There are simple function updates corresponding to assignments in an imperative programming language, which in turn can be composed sequentially and used to form parallel or quantified updates. Updates play a central role in KeY: the verification calculus transforms Java Card programs into updates. KeY contains a powerful and efficient mechanism for simplifying updates and applying them to formulas.

Rule Formalisation and Application. The KeY system has an automated-proof-search mode and an interactive mode. The user can easily switch modes during the construction of a proof.

For interactive rule application, the KeY prover has an easy to use graphical user interface that is built around the idea of direct manipulation. To apply a rule, the user first selects a *focus of application* by highlighting a (sub-)formula or a (sub-)term in the goal sequent. The prover then offers a choice of rules applicable at this focus. This choice remains manageable even for very large rule bases. Rule schema variable instantiations are mostly inferred by matching. A simpler way to apply rules and give instantiations is by drag and drop. If the user drags an equation onto a term the system will try to rewrite the term with the equation. If the user drags a term onto a quantifier the system will try to instantiate the quantifier with this term.

The interaction style is closely related to the way rules are formalised in the KeY prover. There are no hard-coded rules; all rules are defined in the “taclet language” instead. Besides the conventional declarative semantics, taclets have a clear operational semantics, as the following example shows—a “modus ponens” rule in textbook notation (left) and as a taclet (right):

$$\frac{\phi, \psi, \Gamma \Rightarrow \Delta}{\phi, \phi \rightarrow \psi, \Gamma \Rightarrow \Delta} \quad \begin{array}{ll} \backslash \text{find (p} \rightarrow \text{q} \Rightarrow \text{)} & // \textit{implication in antecedent} \\ \backslash \text{assumes (p} \Rightarrow \text{)} & // \textit{side condition} \\ \backslash \text{replacewith(q} \Rightarrow \text{)} & // \textit{action on focus} \\ \backslash \text{heuristics(simplify)} & // \textit{strategy information} \end{array}$$

The `find` clause specifies the potential application focus. The taclet will be offered to the user on selecting a matching focus and if a formula mentioned in the `assumes` clause is present in the sequent. The action clauses `replacewith` and `add` allow modifying (or deleting) the formula in focus, as well as adding additional formulas (not present here). The `heuristics` clause records information for the parameterised automated proof search strategy.

The taclet language is quickly mastered and makes the rule base easy to maintain and extend. Taclets can be proven correct against a set of base taclets [4]. A full account of the taclet language is given in [3] Chapt. 4 and Appendix B.3.3].

Applications. Among the major achievements using KeY in the field of program verification so far are the treatment of the Demoney case study, an electronic purse application provided by Trusted Logic S.A., and the verification of a Java implementation of the Schorr-Waite graph marking algorithm. This algorithm, originally developed for garbage collectors, has recently become a popular benchmark for program verification tools. Chapters 14 and 15 of the KeY book [3] are devoted to a detailed description of these case studies. A case study [14] performed within the HIJA project has verified the lateral module of the flight management system, a part of the on-board control software from Thales Avionics.

Lately we have applied the KeY system also on topics in security analysis [7], and in the area of model-based test case generation [2,10] where, in particular, the prover is used to compute path conditions and to identify infeasible paths.

The flexibility of KeY w.r.t. the used logic and calculus manifests itself in the fact that the prover has been chosen as a reasoning engine for a variety of other purposes. These include the mechanisation of a logic for Abstract State Machines [17] and the implementation of a calculus for simplifying OCL constraints [13].

KeY is also very useful for teaching logic, deduction, and formal methods. Its graphical user interface makes KeY easy to use for students. They can step through proofs with different degrees of automation (using the full verification calculus or just the first-order core rules). The authors have been successfully teaching courses for several years using the KeY system. An overview and course material is available at www.key-project.org/teaching.

Related Tools. There exist a number of other verification systems for object-oriented programs. The KIV¹ tool [1] is closest to ours in that it is also interactive and also based on Dynamic Logic. Most other systems are based on a verification condition generator (VCG) architecture and separate the translation of programs into logic from the actual proof process. A very popular tool of this kind is ESC/Java2² (Extended Static Checker for Java2) [12], which uses the Simplify theorem prover [8] and attempts to find run-time errors in JML-annotated Java programs. ESC/Java2 compromises on completeness and even soundness for the sake of ease of use and scalability. Further systems are JACK [5], Krakatoa [11], LOOP [18], which can also generate verification conditions in higher order logic that may then be proved using interactive theorem provers like PVS, Coq, Isabel, etc. Like KeY, JACK, Krakatoa, and LOOP support JML specifications. With JACK, we moreover share the focus on smart card applications.

3 Verification Case Study: A Calendar Using Interval Trees

In this tutorial, we use a small Java calendar application to illustrate how specifications are written and programs are verified with the KeY system. The application provides typical functionality like creating new calendars, adding or removing appointments, notification services that inform about changes to a particular appointment or a calendar, and views for displaying a time period (like a particular day or month) or for more advanced lookup capabilities.

The class structure of the calendar application is shown in Fig. 1 and 2. It consists of two main packages: a datastructure layer `intervals`, that provides classes for working with (multisets of) intervals, and a domain layer `calendar`, that defines the actual logic of a calendar. Intervals (interface `Interval`) are the basic entities that our calendars are built upon. In an abstract sense, each entry or appointment in a calendar is primarily an interval spanned by its start and its end point in time. A calendar is a multiset of such intervals. For reasons of simplicity, we represent discrete points of time as integers, similarly to the time representation in Unix (the actual unit and offset are irrelevant here). Further, we use the *observer design pattern* (package `observerPattern`) for being able to observe all modifications that occur in a calendar entry.

Interval Datastructures. The most important lookup functionality that our calendar provides, is the ability to retrieve all entries that overlap a certain query time interval (i.e., have a point of time in common with the query interval). Such queries are used, for instance, when displaying all appointments for a particular day. We consequently store intervals in an *interval tree* datastructure [6] (class

¹ www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/

² <http://secure.ucd.ie/products/opensource/ESCJava2/>

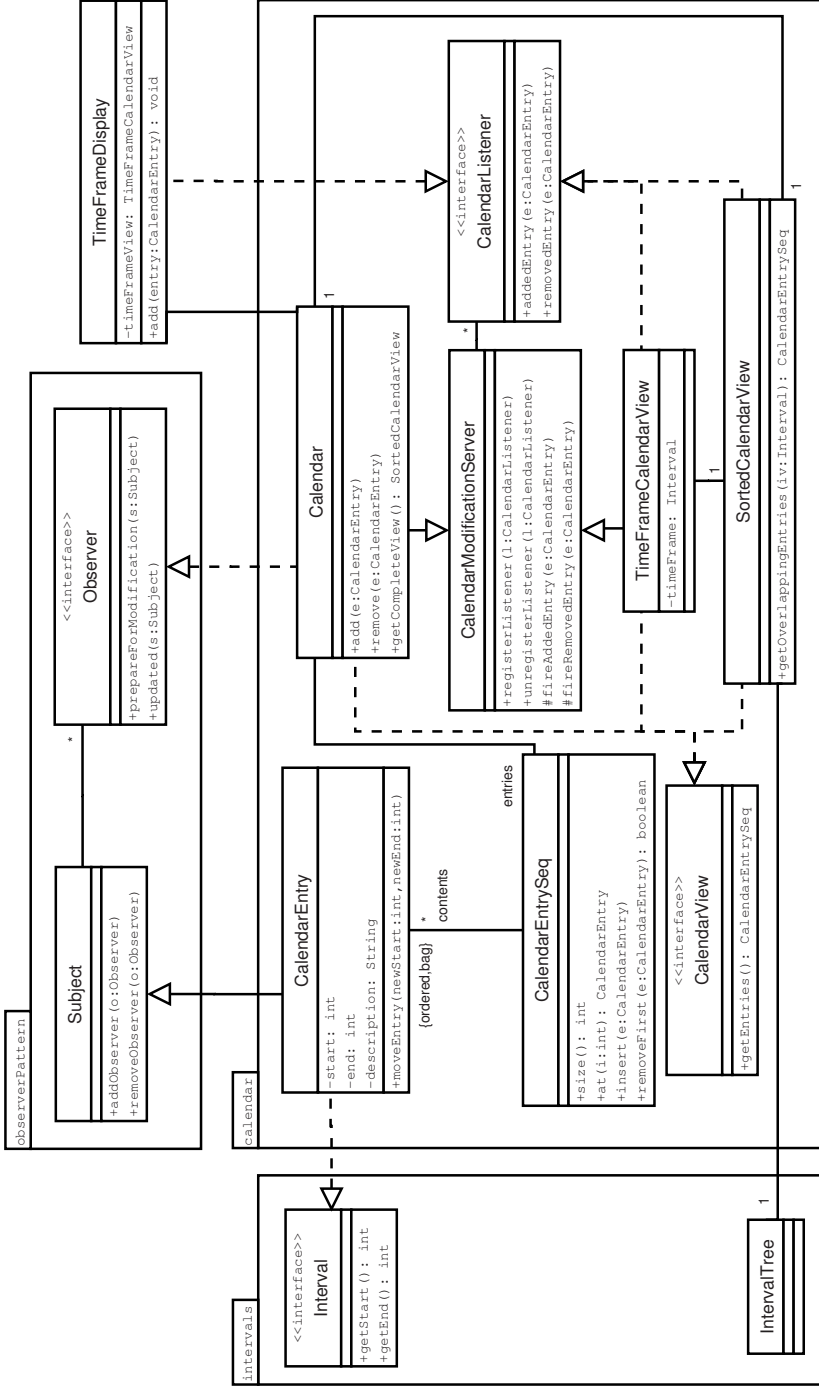


Fig. 1. The packages observerPattern and calendar of the calendar case study

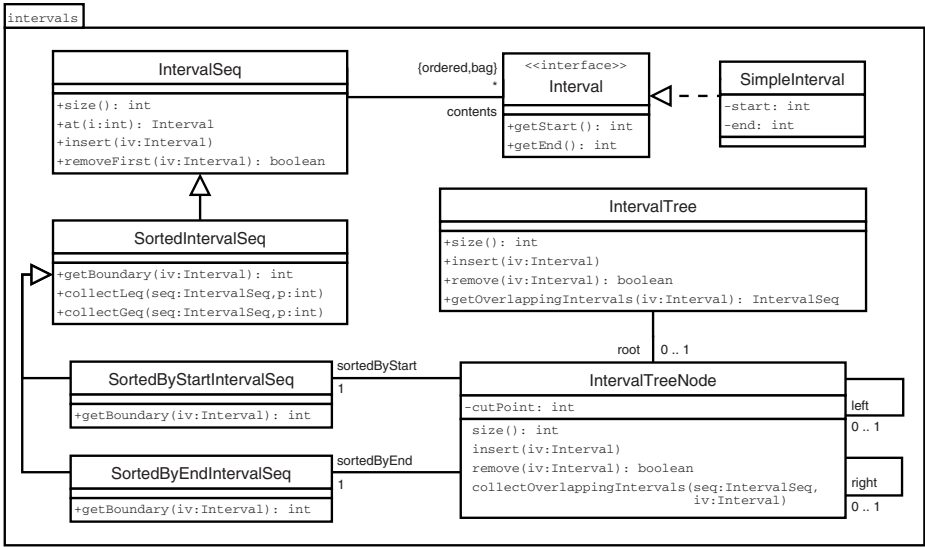


Fig. 2. The package `intervals` of the calendar case study

`IntervalTree` in Fig. 2), which allows to retrieve overlapping entries with logarithmic complexity in the size of the calendar. An interval tree is a binary tree, in which each node (class `IntervalTreeNode`) stores (a) the multiset of intervals that include a certain point (the `cutPoint`) and (b) pointers to the subtrees that handle the intervals strictly smaller (association `left`) resp. strictly bigger (association `right`) than the cut point. The intervals belonging to a particular node have to be stored both sorted by the start and by the end point, which is further discussed in Sect. 4.

Package Calendar. The two primary classes that implement a calendar are `CalendarEntry` for single appointments (an implementation of the interface `Interval`) and `Calendar` for whole calendars. The basic `Calendar` provides the interface `CalendarView` for accessing all entries that are part of the calendar in an unspecified order. A more advanced lookup interface, `SortedCalendarView`, can be accessed through the method `getCompleteView` of `Calendar`. It allows to retrieve all entries that overlap with a given interval. This interface is realised using the interval trees from package `intervals`.

A further view on calendars is `TimeFrameCalendarView`, which pre-selects all appointments within a given period of time, and which is based on the class `SortedCalendarView`. Both `Calendar` and `TimeFrameCalendarView` also provide a notification service (`CalendarModificationServer`) that informs about newly added and removed entries. We illustrate the usage of this service (and of `TimeFrameCalendarView`) in class `TimeFrameDisplay`, which is further discussed and verified in Sect. 5.

4 First Walk-Through: Verifying Insertion into Interval Sequences

In this section, we zoom into a small part of the scenario described above, namely the `insert()` method belonging to the class `IntervalSeq` and its subclasses. In the context of that method, we demonstrate the different basic stages of formal software development with the KeY system. We discuss the *formal specification* of the `insert()` method, the generation of corresponding *proof obligations* in the used program logic, and the *formal verification* with the KeY prover. Along with demonstrating the basic work-flow, we introduce the used formalisms on the way, when they appear, but just to the extent which allows to follow the example. These formalisms are: the specification language JML (Java Modeling Language) [16], the program logic Java Card DL, and the corresponding calculus [3].

As described in Section 3, the basic data structure of the case study scenario is a tree, the nodes of which are instances of the class `IntervalTreeNode`. Each such node contains one integer number (representing a point in time, the “cut point” of the node), and two interval sequences, both containing the same intervals (all of which contain the cut point of the node). The difference between the two sequences is that the contained intervals are sorted differently, once by their start, and once by their end.

Correspondingly, the two sequences contained in each node are instances of the classes `SortedByStartIntervalSeq` and `SortedByEndIntervalSeq`, respectively. Both are subclasses of `SortedIntervalSeq`, which in turn is a subclass of `IntervalSeq`. One of the basic methods provided by (instances of) these classes is `insert(Interval iv)`. In this section, we discuss, as an example, the implementation and specification of that method, as well as the verification of corresponding proof obligations.

The specification of the `insert()` method of the class `SortedIntervalSeq` also involves the superclass, `IntervalSeq`, because parts of the specification are *inherited* from there. Later, in the verification we will also be concerned with the two subclasses of `SortedIntervalSeq`, which provide different implementations of a method called by `insert()`, namely `getBoundary()`.

4.1 Formal Specification and Implementation

Within the Class `IntervalSeq`. This class is the topmost one in this small hierarchy, instances of which represent a sequence of intervals. Internally, the sequence is realised via an array `contents` of type `Interval []`. This array can be longer than the actual `size` of the interval sequence. Thereby, we avoid having to allocate a new array at each and every increase of the sequence’s `size`. Instead, `size` points to the index up to which we consider `contents` be filled with “real” intervals; only if `size` exceeds `contents.length`, a new array is allocated, into

³ All these are described in more detail in the KeY book [3]: JML in Section 5.3, Java Card DL and the calculus in Chapter 3.

which the old one is copied. This case distinction is encapsulated in the method `incSize()`, to be called by `insert()`.

— Java (1.1) —

```
protected void incSize() {
    ++size;
    if ( size > contents.length ) {
        final Interval[] oldAr = contents;
        contents = new Interval[contents.length * 2];
        int i = 0;
        while ( i < oldAr.length ) {contents[i] = oldAr[i]; ++i;}
    }
}
```

— Java —

We turn to the actual `insert()` method now. The class `IntervalSeq` is ignorant of sorting, so all we require from `insert(iv)` is that `iv` is indeed inserted, *wherever*, in the sequence. To the very least, this means that, in a post state, `iv` is stored at *any* of the indices of `contents`. Using mathematical standard notation, we can write this as

$$\exists i. 0 \leq i \wedge i < \text{size} \wedge \text{contents}[i] = \text{iv}$$

Note that, already in this mathematical notation, we are mixing in elements from the programming language level, namely the instance field names, and the array access operator “[]”. Now, the specification language JML takes this several steps further, using Java(like) syntax wherever possible: `<=` for \leq , `&&` for \wedge , `==` for $=$, `!=` for \neq , and so on. Special keywords are provided for concepts not covered by Java, like `\exists` for \exists . Altogether, the above formula is expressed in JML as:

```
\exists int i; 0 <= i && i < size; contents[i] == iv
```

As we can see, quantified formulas in JML have three parts, separated by “;”. The first declares the type of the quantified variable, the second is intended to further restrict the range of the variable, while the third states the “main” property, intuitively speaking. Logically, however, the second and the third part of a JML “`\exists`”-formula are connected via “and” (\wedge).

The above formula is a *postcondition*, as it constrains the admissible states after execution of `insert()`. A sensible *precondition* would be that the interval to be inserted is defined, i.e., not `null`.

Even if we will later expand on the postcondition, we show already how the formulas we have so far can syntactically be glued together, in order to form a JML specification of `insert()`. In general, JML specifications are written into Java source code files, in form of Java comments starting with the symbol “@”. In case of pre/post-conditions, these comments precede the method they specify. In our example, `IntervalSeq.java` would contain the following lines:

 — Java + JML (1.2) —

```

/*@ public normal_behavior
   @ requires iv != null;
   @ ensures (\exists int i; 0 <= i && i < size;
             contents[i] == iv);
   @*/
public void insert(Interval iv) {
    ...

```

 — Java + JML —

This is an example for a *method contract* in JML. For the purpose of our example, this contract is, however, still very weak. It does, for instance, not specify how the values of `size` before and after execution of `insert()` relate to each other. For such a purpose, JML offers the “`\old`” construct, which is used in a postcondition to refer back to the pre-state. With that, we can state `size == \old(size) + 1`. Further, the contract does not yet tell whether all (or, in fact, any) of the intervals previously contained in `contents` remain therein, not to speak of the indices under which they appear. What we need to say is (a) that, up to the index `i` where `iv` is inserted, the elements of `contents` are left untouched, and (b) that all other elements are shifted by one index. Both can be expressed using the universal quantifier in JML, “`\forall`”, which is quite analogous to the “`\exists`” operator. Using that, (b) would translate to:

```

\forall int k; i < k && k < size;
    contents[k] == \old(contents[k-1])

```

Note that, in case of “`\forall`”, the second “`;`” logically is an implication, not a conjunction as was the case for “`\exists`”. In the above formula, `i` refers to the index of insertion, which we have existentially quantified over earlier, meaning we get a nested quantification here.

Together with an appropriate `assignable` clause to be explained below, we now arrive at the following JML specification of `insert()`:

 — Java + JML (1.3) —

```

/*@ public normal_behavior
   @ requires iv != null;
   @ ensures size == \old(size) + 1;
   @ ensures (\exists int i; 0 <= i && i < size;
             contents[i] == iv
             && (\forall int j; 0 <= j && j < i;
                 contents[j] == \old(contents[j]))
             && (\forall int k; i < k && k < size;
                 contents[k] == \old(contents[k-1])));
   @ assignable contents, contents[*], size;
   @*/
public void insert(Interval iv) {
    ...

```

 — Java + JML —

The `assignable` clause, in this example, says that the `insert()` is allowed to change the value of `contents`, the value of the element locations of `contents`, and of `size`, *but nothing else*. The purpose of the `assignable` clauses is not so much the verification of the method `insert` (in this case), but rather to keep feasible the verification of other methods calling `insert()`.

Within the Abstract Class `SortedIntervalSeq`. This class extends the class `IntervalSeq`, augmenting it with the notion of *sortedness*. In particular, this class' implementation of `insert()` must respect the sorting. To specify this requirement in JML, one could be tempted to add sortedness to both, the pre- and the postcondition of `insert()`. However, such invariant properties should rather be placed in JML *class invariants*, which like method contracts are added as comments to the source code.

The following lines are put *anywhere* within the class `SortedIntervalSeq`:

— JML (1.4) —

```

/*@ public invariant
   @   (\forall int i; 0 <= i && i < size - 1;
   @   getBoundary(contents[i]) <= getBoundary(contents[i+1]));
   @*/

```

JML —

The actual sorting criterion, `getBoundary()`, is left to subclasses of this class, by making it an `abstract` method.

— Java + JML (1.5) —

```

protected /*@ pure @*/ abstract int getBoundary(Interval iv);

```

Java + JML —

The phrase “`/*@ pure @*/`” is another piece of JML specification, stating that all implementations of this method terminate (on all inputs), and are free of side effects. Without that, we would not be allowed to use `getBoundary()` in the invariant above, nor in any other JML formula.

Finally, we give the `SortedIntervalSeq` implementation of `insert()` (overriding some non-sorted implementation from `IntervalSeq`):

— Java (1.6) —

```

public void insert(Interval iv) {
    int i = size;
    incSize ();
    final int ivBoundary = getBoundary( iv );
    while ( i > 0 && ivBoundary < getBoundary( contents[i-1] ) ) {
        contents[i] = contents[i - 1];
        --i;
    }
    contents[i] = iv;
}

```

Java —

Within the SortedByStart... and SortedByEnd... Classes. These two classes extend `SortedIntervalSeq` by defining the sorting criteria to be the “start” resp. “end” of the interval. Within `SortedByStartIntervalSeq`, we have:

— Java + JML (1.7) —

```
protected /*@ pure @*/ int getBoundary(Interval iv) {
    return iv.getStart ();
}
```

— Java + JML —


and within `SortedByEndIntervalSeq`, we have

— Java + JML (1.8) —

```
protected /*@ pure @*/ int getBoundary(Interval iv) {
    return iv.getEnd ();
}
```

— Java + JML —

4.2 Dynamic Logic and Proof Obligations

After having completed the specification as described in the previous section we start `bin/runProver` (in your KeY installation directory) as a first step towards verification. The graphical user interface of the KeY prover will pop up. To load files with Java source code and JML specifications, we select `File → Load ...` (or  in the tool bar). For the purposes of this introduction we navigate to where the `calender-sources`⁴ are stored locally, select that very directory (not any of the sub-directories), and push the `open` button. After an instant the JML specification browser will appear on the screen. In the left-most of its three window panes, the `Classes` pane, we expand the folder corresponding to the package `intervals` and select the class `IntervalSeq`. The `Methods` pane now shows all methods of class `IntervalSeq`. We select `void insert(Interval iv)`. Now also the `Proof Obligations` pane shows some entries. We select first the specification case `normal_behavior` and push `Load Proof Obligation`. This brings us back to the KeY prover interface.

Now, the `Tasks` pane records the tasks we have loaded (currently one) and the main window `Current Goal` shows the proof obligation. It looks quite daunting and we use the rest of this section to explain what you see there. The construction of the actual proof is covered in the next section. Ignoring the leading `==>`, the proof obligation is of the form shown in Fig. 3.

Java Card DL. Fig. 3 shows a formula of Dynamic Logic (DL), more precisely Java Card DL, see Section 2. The reader might recognise typical features of first-order logic: the propositional connectives (e.g., `->` and `&`), predicates

⁴ The sources can be downloaded from www.key-project.org/fmco06

KeY

```

1   inReachableState
2   -> \forall intervals.Interval iv_lv;
3     {iv:=iv_lv}
4     (   iv.<created> = TRUE & !iv = null
5         | iv = null
6     -> \forall intervals.IntervalSeq self_IntervalSeq_lv;
7         {self_IntervalSeq:=self_IntervalSeq_lv}
8         ( \forall jint k;
9             _old20(k) = self_IntervalSeq.contents[k - 1]
10        -> \forall jint j; _old19(j) = self_IntervalSeq.contents[j]
11        ->   !self_IntervalSeq = null
12            & self_IntervalSeq.<created> = TRUE
13            & !iv = null
14            & ...
15            & !self_IntervalSeq.contents = null
16            & self_IntervalSeq.contents.length >=
17              self_IntervalSeq.size
18            & self_IntervalSeq.contents.length >= 1
19            & \forall jint i;
20              ( 0 <= i & i < self_IntervalSeq.size
21                -> !self_IntervalSeq.contents[i] = null )
22            & ...
23            & self_IntervalSeq.size >= 0
24        -> {_old17:=self_IntervalSeq.size || _old18_iv:=iv}
25          \<{self_IntervalSeq.insert(iv)@intervals.IntervalSeq;}>
26          (   self_IntervalSeq.size = _old17 + 1
27              & \exists jint i; ... )
28      )
29  )

```

KeY

Fig. 3. Proof obligation for the `insert` method in class `IntervalSeq`

(e.g., `inReachableState`, a predicate of arity 0), equality, constant symbols (e.g., `self_IntervalSeq`), unary function symbols (e.g., `size`), and quantifiers (e.g., `\exists jint i`);). The function symbol `size` is the logical counterpart of the attribute of the same name. Note also that Java Card DL uses dot-notation for function application, for example, `self_IntervalSeq.size` instead of `size(self_IntervalSeq)` on line 17. On line 2, the quantification `\forall intervals.Interval iv_lv`; introduces the quantified variable `iv_lv` of type `Interval` (for disambiguation the package name `intervals` is prefixed). What makes Java Card DL a proper extension of first-order logic are modal operators. In the above example the diamond operator

`\<{self_IntervalSeq.insert(iv)@intervals.IntervalSeq;}>`

occurs on line 25 (note that in KeY the modal operators $\langle \rangle$ and $[\]$ are written with leading backslashes). In general, if $prog$ is any sequence of legal Java Card statements and F is a Java Card DL formula, then $\backslash \langle prog \rangle F$ is a Java Card DL formula too. As already explained in Section 2, the formula $\backslash \langle prog \rangle F$ is true in a state s_1 if there is a state s_2 such that $prog$ terminates in s_2 when started in s_1 and F is true in s_2 . The box operator $\backslash [\dots]$ has the same semantics except that it does not require termination.

In theoretical treatments of Dynamic Logic there is only one kind of variable. In Java Card DL we find it more convenient to separate logical variables (e.g., iv_lv in the above example), from program variables (e.g. iv). Program variables are considered as (non-rigid) constant symbols in Java Card DL and may thus not be quantified over. Logical variables on the other hand are not allowed to occur within modal operators, because they cannot occur as part of Java programs.

State Updates. We are certainly not able to touch on all central points of Java Card DL in this quick introduction, but there is one item we cannot drop, namely *updates*. Let us look at line 3 in Fig. 3. Here $\{iv:=iv_lv\}$ is an example of an update. More precisely, it is an example of a special kind of update, called *function update*. The left-hand side of a function update is typically a program variable, as iv in this example, or an array or field access. The right-hand side can be an arbitrary Java Card DL term, which of course must be compatible with the type of the left-hand expression. Constructs like $\{i:=j++\}$ where the right-hand side would have side-effects are *not* allowed in updates. If $\{lhs:=rhs\}$ is a function update and F is a formula, then $\{lhs:=rhs\}F$ is a Java Card DL formula. The formula $\{lhs:=rhs\}F$ is true in state s_1 if F is true in state s_2 where s_2 is obtained from s_1 by *performing* the update. For example, the state s_2 obtained from s_1 by performing the update $\{iv:=iv_lv\}$ (only) differs in the value of iv , which is in s_2 the value that iv_lv has in s_1 . Java Card DL furthermore allows combinations of updates, e.g., by sequential or parallel composition. An example of such parallel updates, composed via “ \parallel ”, appears in line 24 of Fig. 3.

One difference between updates and Java assignment statements is that logical variables such as iv_lv may occur on the right hand side of updates. In Java Card DL it is not possible to quantify over program variables. This is made up for by the possibility of quantifying over logical variables in updates (as in lines 2 and 3 in Fig. 3). Another role of updates is to store “old” values from the pre-state, that then can be referred to in the post-state (as in lines 24 and 26). Finally, the most important role of updates is that of *delayed substitutions*. During symbolic execution (performed by the prover using the Java Card DL calculus) the effects of a program are removed from the modality $\backslash \langle \dots \rangle$ and turned into updates, where they are *simplified* and *parallelised*. Only when the modality has been eliminated, updates are substituted into the post-state. For a more thorough discussion, we refer to the KeY book [3], Chapt. 3].

Kripke Semantics. A state $s \in S$ contains all information necessary to describe the complete snapshot of a computation: the existing instances of all types, the

values of instance fields and local program variables etc. Modal logic expressions are not evaluated relative to one state model but relative to a collection of those, called a *Kripke Structure*. There are *rigid* symbols that evaluate to one constant meaning in all states of a *Kripke Structure*. The type `jint` (see e.g., line 8 in Fig. 3) in all states evaluates to the (infinite) set of integers, also addition `+` on `jint` are always evaluated as the usual mathematical addition. Logical variables also count among the rigid symbols, no program may change their value. On the other hand there are *non-rigid* symbols like `self_IntervalSeq`, `iv`, `contents`, or `at(i)`.

Proof Obligations. We have to add more details on Java Card DL as we go along but we are now well prepared to talk about proof obligations. We are still looking at Fig. 3 containing the proof obligation in the **Current Goal** pane that was generated by selecting the `normal_behaviour` specification case. Line 11 contains the `requires` clause from the JML method specification for `insert`, while the conjunction of all JML invariants for class `IntervalSeq` appears in lines 15 to 21. If you also need invariants of superclasses or classes that occur as the type of a field in `IntervalSeq` you would tick the *use all applicable invariants* box in the JML browser before generating the proof obligation. Since in the JML semantics `normal_behaviour` includes the termination requirement, the diamond modality is used. Starting with line 26, the first line within the scope of the modal operator, follows the conjunction of the `ensures` clauses in the JML method specification. It is implicit in the JML specification that the above implication should be true for all values of the method parameter `iv` and all instances of class `IntervalSeq` as calling object. As we have already observed this is accomplished in Java Card DL by universal quantification of the logical variables `iv_lv` and `self_IntervalSeq_lv` and *binding* these values to the program variables via the updates `{iv := iv_lv}` and `{self_IntervalSeq := self_IntervalSeq_lv}`. Looking again at Fig. 3, we notice in lines 4 and 5 additional restrictions on the implicitly, via `iv_lv`, universally quantified parameter `iv`. To understand what we see here, it is necessary to explain how Java Card DL handles object creation. Instead of adding a new element to the target state s_2 of a statement `prog` that contains a call to the `new` method we adopt what is called the *constant domain assumption* in modal logic theory. According to this assumption all states share the same objects for all occurring types. In addition there is an implicit field `<created>` of the class `java.lang.Object` (to emphasise that this is not a normal field, it is set within angled brackets). Initially we have `o.<created> = FALSE` for all objects `o`. If a `new` method call is performed we look for the next object `o` to be created and change the value of `o.<created>` from `FALSE` to `TRUE`, which now is nothing more than any other function update.

Pre-Values of Arrays. It is quite easy to track the preconditions occurring in the Java Card DL proof obligation to their JML origin as a `requires` or `invariant` clause. But, there are also parts in the Java Card DL precondition that do not apparently correspond to a JML clause, for example in Fig. 3 line 10:

```
\forallall jint j; _old19(j) = self_IntervalSeq.contents[j]
```

This is triggered by the use of the `\old` construct in the following part of the JML ensures clause:

— JML —

```
@   && (\forall int j; 0 <= j && j < i;
@       contents[j] == \old(contents[j]))
```

— JML —

KeY handles this by introducing a new function, here `_old19`, and requiring it to coincide for all arguments with `self_IntervalSeq.contents`. Since `_old19` is not affected by the program it can be used after the diamond operator to refer to the old values of `self_IntervalSeq.contents` as in:

```
\forall int jint j; (0 <= j & j < i ->
    self_IntervalSeq.contents[j] = _old19(j))
```

Imagine that you were to write a run-time checker for JML. That will give you the idea for how you would implement the `\old` construct. Corresponding to the JML term `\old(size)` the definition `_old17:=self_IntervalSeq.size` is introduced. This time not as an equality, but as already mentioned above as an update. The advantage is that the update will be automatically applied to prove the postcondition `size = \old(size) + 1`. The application of the definition of `_old19(j)` on the other hand requires user interaction or special heuristics to pick the needed instantiations of `j`.

Proper Java States. It still remains to comment on the precondition that we skipped on first reading, `inReachableState`. In KeY a method contract is proved by showing that the method terminates in a state satisfying the postcondition when started in any state s_1 satisfying the preconditions and the invariants. This may also include states s_1 that cannot be reached from the `main` method. But, usually the preconditions and invariants narrow down this possibility and in the end it does not hurt much to prove a bit more than is needed. But, there is another problem here: the implicit fields. A state with object `o` and field `a` such that `o.<created> = TRUE, o.a != null, and o.a.<created> = FALSE` is not possible in Java, but could be produced via updates. It is the precondition `inReachableState` that excludes this kind of anomalies.

Capturing JML Specifications in Java Card DL. Let us go back to the JML specification browser and select `Assignable P0` for the `insert` method. When proving, e.g., the `normalbehaviour` clause of a method contract, we also take advantage of the JML `assignable` clause. The current proof obligation now checks if the `assignable` clauses are indeed correct. In the case at hand there does not seem to be much to do, since all fields of class `IntervalSeq` (there are just two) may change. To be precise, a call to the `insert` method only assigns to these fields for the calling object, otherwise they should remain unchanged. This is what this proof obligation states.

Now, let us select the last proof obligation in the JML specification browser, which is named `class specification`. Its purpose is to make sure that, for

any state s_1 that satisfies all invariants of the `IntervalSeq` class and the preconditions of `insert(iv)`, the invariants are again true in the end state s_2 of this method. Note that here the modal box operator is used. Termination of the method was already part of its method contract, so we need not prove it again here. The proof obligation requires the invariants to also hold when the methods terminates via an exception. This is the reason why `insert(iv)` is enclosed in a `try-catch` block. Also the `inReachableState` predicate is among the invariants to be proved.

4.3 Verification

In this section, we demonstrate how the KeY prover is used to verify a proof obligation resulting from our example. It is important to note, however, that a systematic introduction into the usage of the prover is beyond the scope of this paper. Such an introduction can be found in Chapter 10 of the KeY book [3]. On the other hand, the examples in that chapter are of toy size as compared to the more realistic proof obligations we consider in this paper.

This section is meant to be read with the KeY prover up and running, to perform the described steps with the system right away. The exposition aims at giving an *impression* only, on how verification of more realistic examples is performed, while we cannot explain in detail *why* we are doing what we are doing. Again, please refer to [3, Chapt. 10] instead.

We will now verify that the implementation of the method `insert()` in class `SortedIntervalSeq` (not in `IntervalSeq`) respects the contract that it inherits from `IntervalSeq`. Before starting the proof, we remind ourselves of the code we are going to verify: the implementation of `insert()` was given in listing (1.6) in Sect. 4.1, and it calls the inherited method `incSize()`, see listing (1.1). Both these methods contain one `while` loop, which we advise the reader to look at, as we have to recognise them at some point during the verification.

We first let KeY generate the corresponding proof obligation, by following the same steps as described at the beginning of Sect. 4.2 (from `File` \rightarrow `Load ...` onwards), but with the difference that this time we select, in the `Classes` pane, `SortedIntervalSeq` instead of `IntervalSeq`. (Apart from that, it is again the method `void insert(Interval iv)` that we select in the `Methods` pane, and the specification case `normal_behavior` that we select in the `Proof Obligations` pane. Before clicking on the `Load Proof Obligation` button, we make sure that the two check-boxes at the bottom of the specification browser are *not* checked.)

Afterwards, the `Current Goal` pane contains a proof obligation that is very similar to the one discussed in Sect. 4.2, just that now the (translated) class invariant of `SortedIntervalSeq`, see listing (1.4), serves as an additional assumption.

This now is a good time to comment on the the leading “ \Rightarrow ” symbol in the `Current Goal` pane. As described in Sect. 2, the KeY prover builds proofs based on a *sequent calculus*. Sequents are of the form $\phi_1, \dots, \phi_n \Rightarrow \phi'_1, \dots, \phi'_m$, where ϕ_1, \dots, ϕ_n and ϕ'_1, \dots, ϕ'_m are two (possibly empty) comma-separated lists of formulas, separated by the sequent arrow \Rightarrow (that is written as “ \Rightarrow ” in the KeY system). The intuitive meaning of a sequent is: if we assume all formulas

ϕ_1, \dots, ϕ_n to hold, then *at least one* of the formulas ϕ'_1, \dots, ϕ'_m holds. We refer to “ ϕ_1, \dots, ϕ_n ” and “ ϕ'_1, \dots, ϕ'_m ” as the “left-hand side” (or “antecedent”) and “right-hand side” (or “succedent”) of the sequent, respectively.

The particular sequent we see now in the **Current Goal** pane has only one formula on the right-hand side, and no formulas on the left-hand side, which is the typical shape for generated proof obligations, prior to application of any calculus rule. It is the purpose of the sequent calculus to, step by step, take such formulas apart, while collecting assumptions on the left-hand side, and alternatives on the right-hand side, until the sheer shape of a sequent makes it trivially true. Meanwhile, certain rules make the proof branch.

We prove this goal with the highest possible degree of automation. However, we first apply one rule interactively, just to show how that is done. In general, interactive rule application is supported by the system offering only those rules which are applicable to the highlighted formula, resp. term (or, more precisely, to its top-level operator). If we now click on the leading “ \rightarrow ” of the right-hand side formula, a context menu for rule selection appears. It offers several rules applicable to “ \rightarrow ”, among them **imp_right**, which in textbook notation looks like this:

$$\text{imp_right} \quad \frac{\Gamma, \phi \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \phi \rightarrow \psi, \Delta}$$

A tool-tip shows the corresponding taclet. Clicking on **imp_right** will apply the rule in our proof, and the **Current Goal** pane displays the new goal. Moreover, the **Proof** tab in the lower left corner displays the structure of the (unfinished) proof. The nodes are labelled either by the name of the rule which was applied to that node, or by “OPEN GOAL” in case of a goal. (In case of several goals, the one currently in focus is highlighted in blue.) We can see that **impRight** has been applied *interactively* (indicated by a hand symbol), and that afterwards, **Update Simplification** has been applied automatically. Updates are simplified automatically after each interactive step, because there is usually no reason to work with formulas that contain unsimplified updates.

The proof we are constructing will be several thousand steps big, so we better switch to automated proof construction now. For that, we select the **Proof Search Strategy** tab in the lower left corner, and configure the proof strategy as follows:

- **Max. rule applications: 5000** (or just any big number)
- **Java DL** (the strategy for proving in JavaDL)
- **Loop treatment: None** (we want symbolic execution to stop in front of loops)
- **Method treatment: Expand** (methods are inlined during symbolic execution)
- **Query treatment: Expand** (queries in specifications are inlined)
- **Arithmetic treatment: Basic** (simple automatic handling of linear arithmetic)
- **Quantifier treatment: Non-Splitting Instantiation**
(use heuristics for automatic quantifier handling, but do not perform instantiations that might cause proof splitting)

We run the strategy by clicking the ► button (either in the **Proof Search Strategy** tab or in the tool bar). The strategy will stop after about 1000 rule applications once the symbolic execution arrives at loops in the program (due to **Loop treatment: None**). We open the **Goals** tab, where we can see that there are currently five goals left to be proven.

We can view each of these goals in the **Current Goal** pane, by selecting one after the other in the **Goals** tab. In four of the five goals, the modality (preceded by a parallel update) starts with:

— KeY (1.9) —

```
\<{method-frame(...): {
    while ( i>0 && ivBoundary<getBoundary(contents[i-1]) ) {
        ...
    }
```

— KeY —

In those four proof branches, symbolic execution is just about to “enter” the while loop in the method `insert()`. In the remaining fifth branch, the same holds for the while loop in the method `incSize()`. For the loop in `insert()`, we get four cases due to the two existing implementations of the interface `Interval` and the two concrete subclasses of the abstract class `SortedIntervalSeq`. All four cases can be handled in the same way and by performing the same interactions, to be described in the following.

In each case, we first have to process the while loop at the beginning of the modality. It is well known that loops cannot be handled in a similarly automated fashion as most other constructs.

Loop Invariants. Generally, for programs containing loops we have to choose a suitable *loop invariant*⁵ (a formula) in order to prove that the loop has the desired effect and a *loop variant* (an integer term) for proving that the loop terminates. We also have to specify the *assignable memory locations* that can be altered during execution of the loop. All this information can be entered as part of an interactive proof step in KeY. However, the prover also supports the JML feature of annotating loops with invariants, variants, and assignable locations.

Invariants typically express that the loop counter is in a valid range, and give a closed description of the effect of the first n iterations. For the loop in the method `insert()`, it is necessary to state in the invariant that:

- the loop counter `i` never leaves the interval $[0, \text{size})$,
- the interval is not inserted too far left in the array, and
- the original contents of the sequence are properly shifted to the right.

The last component of the invariant is very similar to the post-condition, see listing (L.3), of the whole `insert()` method, which expresses that the argument of the method is properly inserted in the sequent (at an arbitrary place). In JML, we could capture this part of the loop invariant as:

⁵ As an alternative to using invariants, KeY offers induction, see [3] Chapt. 11].

— JML —

```
(\forallall int k; 0 <= k && k < i;
  contents[k] == \old(contents[k])) &&
(\forallall int k; i < k && k < size;
  contents[k] == \old(contents[k-1]))
```

— JML —

In this constraint, *i* is the loop variable. It is stated that all array components that have already been visited by the loop are shifted to the right, whereas a prefix of the array remains unchanged.

Due to a certain shortcoming in the current JML support within KeY, however, it is for the time being not possible to use the `\old` operator in JML loop invariants. (It will be possible in future versions of KeY.) We can work around this problem by introducing a *ghost field* (a specification-only variable) that stores the old contents of the array. This is done by adding the following lines to the class `IntervalSeq`:

— Java + JML —

```
/*@ public ghost Interval[] oldContents;

/*@ public normal_behavior
  @ ...
  @ requires contents != oldContents && oldContents != null
  @           && oldContents.length == contents.length
  @           && (\forallall int i; 0 <= i && i < size;
  @               contents[i] == oldContents[i]);
  @*/
public void insert(Interval iv) {
  ...
```

— Java + JML —

We can then write the complete loop invariant as shown below. Stating the termination of the loop is simple, because the variable *i* is always non-negative and decreased in each iteration. Further, we specify that the only modifiable memory locations are the loop counter and the elements of the array `contents`.

— Java + JML —

```
/*@ loop_invariant 0 <= i && i < size &&
  @   (i < size - 1 ==>
  @   ivBoundary < getBoundary( contents[i] )) &&
  @   (\forallall int k; 0 <= k && k < i;
  @       contents[k] == oldContents[k]) &&
  @   (\forallall int k; i < k && k < size;
  @       contents[k] == oldContents[k-1]);
  @ decreases i;
  @ assignable contents[*], i;
  @*/
```

```

while ( i > 0 && ivBoundary < getBoundary( contents[i-1] ) ) {
    contents[i] = contents[i - 1];
    --i;
}

```

Java + JML

Verification Using Invariants. We continue our proof on one of the four similar goals, all of which containing the modality of the form (L.9), such that processing the loop in `insert()` is the next step. Because all these four goals can be handled in the same way, we can pick an arbitrary one of them, by selecting it in the **Goals** tab. Before proceeding, we switch to the **Proof** tab, to better see the effect of the upcoming proof step.

We apply an invariant rule which automatically extracts the JML annotation of our loop from the source code. For that, we click on any of the “:=” symbols in the parallel update preceding the modality $\langle \dots \rangle$, and select `while_invariant_with_variant_dec` from the rules offered. A **Choose Tactlet Instantiation** window pops up, where we just press the **Apply** button.

Afterwards, the **Proof** tab tells us (possibly after scrolling down a bit) that the application of this invariant rule has resulted in four proof branches:

- **Invariant Initially Valid:** It has to be shown that the chosen invariant holds when entering the loop.
- **Body Preserves Invariant:** Under the assumption that the invariant and the loop condition hold, after one loop iteration the invariant still has to be true.
- **Termination:** Under the assumption that the invariant and the loop condition hold, the chosen variant has to be decreased by the loop body, but has to stay non-negative.
- **Use Case:** The remaining program has to be verified now using the fact that after the loop terminates, the invariant is true and the loop condition is false.

The four cases can be proven as follows. Generally, for a complex proof like this, it is best to handle the proof goals one by one and to start the automatic application of rules only locally for a particular branch. This is done by clicking on a sequent arrow \Rightarrow and choosing **Apply rules automatically here**, or by shift-clicking on a sequent arrow, or by right-clicking on a node in the proof tree display and selecting **Apply Strategy** from the context menu. (Clicking on \blacktriangleright , in contrast, will apply rules to *all* remaining proof goals, which is too coarse-grained if different search strategy settings have to be used for different parts of the proof.)

Also, please note that a proof branch beginning with a green folder symbol is closed. Therefore, this symbol is a success criteria in each of the following four cases. Moreover, branches in the **Proof** tab can be expanded/collapsed by clicking on \boxplus/\boxminus . To keep a better overview, we advise the reader to collapse the branches of the following four cases once they are closed.

Invariant Initially Valid. The proof obligation can easily be handled automatically by KeY and requires about 100 rule applications.

Body Preserves Invariant. This is the goal that requires the biggest (sub-)proof with about 11000 rule applications. In the **Proof Search Strategy** pane, choose a maximum number of rule applications of 20000 and run the prover in auto-mode on the goal as described above. This will, after a while, close the “Body Preserves Invariant” branch.

Termination. Proceed as for the case **Body Preserves Invariant** (possibly after expanding the branch and selecting its **OPEN GOAL**). This case will be closed after about 6000 steps.

Use Case. Proceed similarly as for the case **Body Preserves Invariant**. At first, calling the automated strategy will perform about 5000 rule applications. In contrast to the other three cases, for this last branch it is also necessary to manually provide witnesses for certain existentially quantified formulas (in the succedent) that can neither be found by KeY, nor by the external prover Simplify [9], automatically. These formulas correspond to the post-condition of the method `insert()`, where a point has to be “guessed” at which a further element has been added to the sequence. The form of the formulas is:

$$\exists \text{joint } i; \forall \text{joint } j; \forall \text{joint } k; F$$

Fortunately, for this problem, it is easy to read off the witness `i` that allows to prove the formulas: the body F always contains equations of the form $i = t$, where t is the desired witness. To actually perform the instantiation of the formula, drag the term t to the quantifier `\exists joint i` and choose the rule `ex_right_hide` in the appearing menu [6]. After this instantiation step, call the automated strategy, locally, and in those cases where this does not close that very branch, apply Simplify, *only locally*. A branch local call of Simplify is performed by clicking on the sequent arrow `==>`, and selecting **Decision Procedure Simplify** in the context menu. In this way, handle all branches with formulas of the above form, until the “Use Case” has a green folder at its beginning, meaning this case is closed.

5 Second Walk-Through: Specifying and Verifying Timeframe Displays

In this section, we practice specification and verification a second time, now with higher speed, and coarser granularity. The example is the method `add()` of the class `TimeFrameDisplay`.

5.1 Formal Specification and Implementation

Within the Class `TimeFrameDisplay`. The class `TimeFrameDisplay` is a concrete application of the calendar view `TimeFrameCalendarView`, and could be

⁶ In case the option **Options → DnD Direction Sensitive** is enabled, KeY will perform the instantiation without showing a menu.

(the skeleton of) a dialogue displaying a certain time period in a calendar. On the following pages, we demonstrate how we can give a more behavioural specification for some aspects of such a dialogue. The investigated method is `TimeFrameDisplay::add`, which simply delegates the addition of a new entry to the underlying `Calendar` object:


— Java + JML (1.10) —

```
public class TimeFrameDisplay implements CalendarListener {
    ...
    /*@ public normal_behavior
       @ requires entry != null;
       @ requires overlapping ( timeFrame, entry );
       @ ensures lastEntryAdded == entry;
    @*/
    public void add(CalendarEntry entry) {
        cal.add ( entry );
    }
    ...
    private CalendarEntry lastEntryAdded = null;
    public void addedEntry(CalendarEntry e) {
        lastEntryAdded = e;
    }
}
```

— Java + JML —

In this context, we would like to specify that calling `add` actually results in a new calendar entry being displayed on the screen. In order to simulate this effect, we introduce an attribute `lastEntryAdded` that is assigned in the method `addedEntry`. The post-condition of method `add`, `lastEntryAdded == entry`, consequently states that calling `add` eventually raises the signal `addedEntry` with the right argument (see Fig. 4 for an illustration).

5.2 Proof Obligations and Verification

This time, we demonstrate the use of a hand-written Java Card DL proof obligation instead of importing a JML specification into KeY. Formulating a problem directly in DL is more flexible and gives us full control over which assumptions we want to make, but it is also more low-level, more intricate, and requires more knowledge about the logic and the prover. Figure 5 shows the main parts of the file `timeFrameDisplayAdd.key` containing the proof obligation. A full account on the syntax used in KeY input files is given in [3, Appendix B]. As before, we can load `timeFrameDisplayAdd.key` by selecting `File → Load ...` (or  in the tool bar) and choosing the file in the appearing dialogue.

The KeY input file in Fig. 5 starts with the path to the Java sources under investigation, and with a part that declares a number of program variables

 KeY

```

1  \javaSource "calendar-sources/";
2  \programVariables {
3    calendar.CalendarEntry entry; calendar.CalendarEntry old_entry;
4    TimeFrameDisplay self;
5  \problem {
6    inReachableState
7    & self != null & self.<created> = TRUE
8    & entry != null & entry.<created> = TRUE
9    & TimeFrameDisplay.overlapping(self.timeFrame, entry) = TRUE
10
11   & self.cal!=null & self.timeFrame!=null & self.timeFrameView!=null
12   & self.timeFrameView.listenersNum = 1 & self.cal.listenersNum = 2
13   & self.timeFrameView.listeners[0] = self
14   & self.cal.listeners[0] = self.cal.completeView
15   & self.cal.listeners[1] = self.timeFrameView
16   & self.timeFrameView.timeFrame = self.timeFrame
17
18   & \forallall calendar.CalendarModificationServer serv;
19     ( serv != null & serv.<created> = TRUE
20     -> serv.listeners != null
21         & 0 <= serv.listenersNum & 1 <= serv.listeners.length
22         & serv.listenersNum <= serv.listeners.length)
23   & \forallall observerPattern.Subject subj;
24     ( subj != null & subj.<created> = TRUE
25     -> subj.observers != null
26         & 0 <= subj.observersNum & 1 <= subj.observers.length
27         & subj.observersNum <= subj.observers.length)
28   & \forallall calendar.CalendarEntrySeq entry;
29     ( entry != null & entry.<created> = TRUE
30     -> entry.contents != null
31         & 0 <= entry.size & 1 <= entry.contents.length
32         & entry.size <= entry.contents.length)
33   & \forallall calendar.Calendar cal;
34     ( cal != null & cal.<created> = TRUE
35     -> cal.entries != null & cal.completeView != null)
36   & \forallall calendar.TimeFrameCalendarView view;
37     ( view != null & view.<created> = TRUE
38     -> view.completeView != null & view.timeFrame != null
39         & view.cal != null)
40   & \forallall calendar.SortedCalendarView view;
41     (view != null & view.<created> = TRUE -> view.entryTree != null)
42
43   -> {old_entry := entry} \<{ self.add(entry)@TimeFrameDisplay; }\>
44     self.lastEntryAdded = old_entry }

```

 KeY

Fig. 5. The hand-written proof obligation for Sect. 5

(lines 2–4) used in the specification. The main part of the file describes one particular scenario that we want to simulate:

- In lines 7–8, we assume that `self` and `entry` refer to proper objects of classes `TimeFrameDisplay` resp. `CalendarEntry`. The calendar entry is also supposed to overlap with the attribute `self.timeFrame` (line 9), which is the pre-condition of the method `TimeFrameDisplay::add`.
- The `TimeFrameDisplay` object `self` has been properly set up and connected to a `Calendar` and to a `TimeFrameCalendarView` (lines 11, 16). The freshly created `TimeFrameCalendarView` has exactly one listener attached, namely the object `self` (lines 12, 13). Likewise, the calendar `self.cal` does not have any listeners registered apart from its `SortedCalendarView` and the `TimeFrameCalendarView` (lines 12, 14, 15).
- In order to perform the verification, we need to assume a number of invariants. Lines 18–32 contain three very similar class invariants for the classes `CalendarModificationServer`, `Subject`, and `CalendarEntrySeq`, mostly expressing that the arrays for storing listeners and calendar entries are sufficiently large. In lines 33–41, we state somewhat simpler invariants for `Calendar`, `TimeFrameCalendarView`, and `SortedCalendarView` that ensure that attributes are non-null.

In this setting, we want to show that an invocation of the method `self.add` with parameter `entry` has the effect of raising a signal `addedEntry`. This property is stated in lines 43–44 using a diamond modal operator.

Loop Handling. Apart from sequential code that can simply be executed symbolically, there are three loops in the system that require our attention in this setting. The loops in the methods `Subject::registerObserver` (① in Fig. 4) and `CalendarEntrySeq::incSize` (② in Fig. 4) are similar in shape and are necessary for handling the dynamically growing arrays of entries and listeners:

— Java + JML —

```
public void registerObserver(Observer obs) {
    ++observersNum;
    if ( observersNum > observers.length ) {
        final Observer[] oldAr = observers;
        observers = new Observer[observers.length * 2];
        int i = 0;
        while ( i < oldAr.length ) {observers[i] = oldAr[i]; ++i;}
    }
    observers[observersNum - 1] = obs;
}
...
protected void incSize() {
    ++size;
    if ( size > contents.length ) {
        final CalendarEntry[] oldAr = contents;
```



```

    contents = new CalendarEntry[contents.length * 2];
    int i = 0;
    while ( i < oldAr.length ) {contents[i] = oldAr[i]; ++i;}
  }
}

```

Java + JML

We can handle both loops in the same way (and with the same or similar invariants) as in Sect. 4.3. As before, it is enough to annotate the loops with JML invariants and variants, which can be read and extracted by KeY during the verification.

The third occurrence of a loop is in the class `CalendarModificationServer` in package `calendar` (③ and ⑥ in Fig. 4):

KeY

```

protected void fireAddedEntry(CalendarEntry entry) {
    int i = 0;
    while ( i != listenersNum ) {
        listeners[i].addedEntry ( entry ); ++i;}
}

```

KeY

This loop is executed after adding a new entry to the calendar and is responsible for informing all attached listeners about the new entry. In our particular scenario, there are exactly two listeners (the objects `self.cal.completeView` and `self.timeFrameView`), and therefore we can handle this loop by unwinding it twice.

The Actual Verification, Step by Step. After loading the problem file shown in Fig. 5, we select proof search options as in Sect. 4.3:

- Loop treatment: None
- Method treatment: Expand
- Query treatment: None
 - (we do not inline queries immediately, because we want to keep the expression `TimeFrameDisplay.overlapping(self.timeFrame,entry)` that occurs in Fig. 5 for later)
- Arithmetic treatment: Basic
- Quantifier treatment: Non-splitting instantiation

Running the prover with these options and about 1000 rule applications gets us to the point where we have to handle the loops of the verification problem. There are three goals left, corresponding to the points ①, ② and ③ in Fig. 4, one for each of the loops that are described in the previous paragraph. This is due to the fact that the loops in the methods `incSize` and `registerObserver` are only executed if it is necessary to increase the size of the arrays involved. Consequently, the proof constructed so far contains two case distinctions and

three possible cases. As the loops in `incSize` and `registerObserver` can be eliminated using invariants (exactly as in the previous section), we concentrate on the third loop in method `fireAddedEntry` that is met at point ③ in Fig. 4.

In order to unwind the loop of `fireAddedEntry` once, click on the program block containing the method body and choose the rule `unwind_while`. This duplicated the loop body and guards it with a conditional statement. That is, the loop “`while(b){prog}`” is replaced by “`if(b){prog;while(b){prog}}`”.

After unwinding the loop, we have to deal with the first object listening for changes in the calendar, which is a `SortedCalendarView`. To continue, select `Method treatment: None` and run the prover in automode. The prover will stop at the invocation `SortedCalendarView::addedEntry` (④ in Fig. 4), which we can unfold using the rule `method_body_expand`. After that, continue in automode.

Method Contracts. The method `SortedCalendarView::addedEntry` inserts the new `CalendarEntry` into an interval tree to enable subsequent efficient lookups. Consequently, the next point where the prover stops is an invocation of the method `IntervalTree::insert`. The exact behaviour of this insertion is not important for the present verification problem, however, so we get rid of it using a *method contract* that only specifies which parts of the program state could possibly be affected by the insertion operation. Such a contract can be written based on Dynamic Logic and is shown in Fig. 6 (it is contained in the file `timeFrameDisplayAdd.key`). We specify that the pre-condition of the method `IntervalTree::insert` is `ivt != null & iv != null`, that arbitrary things can hold after execution of the method (the post-condition is `true`), but that only certain attributes of classes in the `intervals` package can be modified (the attributes listed behind the keyword `\modifies`).

In order to apply the method contract, we click on the program and select the item `Use Method Contract` in the context menu. In the appearing dialogue, we have to select the right contract `intervalTreeInsert`. This leads to two new proof goals, in one of which it has to be shown that the pre-condition of the contract holds, and one where the post-condition is assumed and the remaining program has to be handled. By continuing in automode, the first goal can easily be closed, and in the second goal the prover will again stop at point ③ in front of the loop of method `fireAddedEntry` (the second iteration of the loop).

Coming Back to TimeFrameDisplay. The next and last callback that needs to be handled is the invocation of `TimeFrameCalendarView::addedEntry` at point ⑤. This method checks whether the calendar entry at hand overlaps with the time period `TimeFrameCalendarView::timeFrame`, and in this case it will forward the entry to the `TimeFrameDisplay`:

— Java + JML —

```
public void addedEntry(CalendarEntry e) {
    if ( overlapping ( timeFrame, e ) ) fireAddedEntry ( e );
}
```

— Java + JML —

```

— KeY —
\contracts {
  intervalTreeInsert {
    \programVariables {
      intervals.IntervalTree ivt; intervals.Interval iv;
    }
    ivt != null & iv != null
    -> \<{ ivt.insert(iv)@intervals.IntervalTree; }>
    true
    \modifies { ivt.root,
      \for intervals.IntervalTreeNode n; n.cutPoint,
      \for intervals.IntervalTreeNode n; n.left,
      \for intervals.IntervalTreeNode n; n.right,
      \for intervals.IntervalTreeNode n; n.sortedByStart,
      \for intervals.IntervalTreeNode n; n.sortedByEnd,
      \for intervals.IntervalTreeNode n; n.sortedByStart.size,
      \for intervals.IntervalTreeNode n; n.sortedByStart.contents,
      \for (intervals.IntervalTreeNode n; int i)
        n.sortedByStart.contents[i],
      \for intervals.IntervalTreeNode n; n.sortedByEnd.size,
      \for intervals.IntervalTreeNode n; n.sortedByEnd.contents,
      \for (intervals.IntervalTreeNode n; int i)
        n.sortedByEnd.contents[i] }
  };
}

```

KeY

Fig. 6. Java Card DL contract for the method `CalendarEntry::insert`

The property `overlapping(timeFrame, e)` is given as a pre-condition of the method `TimeFrameDisplay::add` and now occurs as an assumption in the antecedent of the goal:

```
TimeFrameDisplay.overlapping(self.timeFrame, entry) = TRUE
```

We can simply continue with symbolic execution on the proof branch. Because we want the prover to take all available information into account and not to stop in front of loops and methods anymore, select **Loop treatment: Expand**, **Method treatment: Expand**, and **Query treatment: Expand**. Choose a maximum number of rule applications of about 5000. Then, click on the sequent arrow `==>` and select **Apply rules automatically here**. This eventually closes the goal.

6 Conclusion

In this paper we walked step-by-step through two main verification tasks of a non-trivial case study using the KeY prover. Many of the problems encountered here—for example, the frame problem, what to include into invariants, how to

modularise proofs—are discussed elsewhere in the research literature, however, typical research papers cannot provide the level of detail that would one enable to actually trace the details. We do not claim that all problems encountered are yet optimally solved in the KeY system, after all, several are the target of active research [15]. What we intended to show is that realistic Java programs actually can be specified and verified in a modern verification system and, moreover, all crucial aspects can be explained within the bounds of a paper while the verification process is to a very large degree automatic. After studying this tutorial, the ambitious reader can complete the remaining verification tasks in the case study.

As for “future work,” our ambition is to be able to write this tutorial without technical explanations on how the verification is done while covering at least as many verification tasks. We would like to treat modularisation and invariant selection neatly on the level of JML, and the selection of proof obligations in the GUI. It should not be necessary anymore to mention Java Card DL in any detail. From failed proof attempts, counter examples should be generated and animated without the necessity to inspect Java Card DL proof trees. There is still some way to go to mature formal software verification into a technology usable in the mainstream of software development.

Acknowledgements

We would like to thank Richard Bubel for many discussions on various topics of the paper, and for his enormous contribution to the constant improvement of the KeY system.

References

1. Balsler, M., Reif, W., Schellhorn, G., Stenzel, K., Thums, A.: Formal system development with KIV. In: Maibaum, T.S.E. (ed.) ETAPS 2000 and FASE 2000. LNCS, vol. 1783, Springer, Heidelberg (2000)
2. Beckert, B., Gladisch, C.: White-box testing by combining deduction-based specification extraction and black-box testing. In: Gurevich, Y. (ed.) Proceedings, Testing and Proofs, Zürich, Switzerland. LNCS, Springer, Heidelberg (2007)
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
4. Beckert, B., Klebanov, V.: Must program verification systems and calculi be verified. In: Proceedings, 3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC), Seattle, USA (2006)
5. Burdy, L., Requet, A., Lanet, J.-L.: Java applet correctness: A developer-oriented approach. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 422–439. Springer, Heidelberg (2003)
6. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. McGraw-Hill Higher Education, New York (2001)
7. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) SPC 2005. LNCS, vol. 3450, pp. 193–209. Springer, Heidelberg (2005)

8. Detlefs, D., Nelson, G., Saxe, J.: Simplify: A Theorem Prover for Program Checking. Technical Report HPL-2003-148, HP Labs (July 2003)
9. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. *Journal of the ACM* 52(3), 365–473 (2005)
10. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: Gurevich, Y. (ed.) *Proceedings, Testing and Proofs*, Zürich, Switzerland. LNCS, Springer, Heidelberg (2007)
11. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) *19th International Conference on Computer Aided Verification*. LNCS, Springer, Berlin, Germany (2007)
12. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, pp. 234–245. ACM Press, New York (2002)
13. Giese, M., Larsson, D.: Simplifying transformations of OCL constraints. In: Briand, L.C., Williams, C. (eds.) *MoDELS 2005*. LNCS, vol. 3713, Springer, Heidelberg (2005)
14. Hunt, J.J., Jenn, E., Leriche, S., Schmitt, P., Tonin, I., Wonnemann, C.: A case study of specification and verification using JML in an avionics application. In: Rochard-Foy, M., Wellings, A. (eds.) *Proc. of the 4th Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, ACM Press, New York (2006)
15. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Technical Report 06-14a, Department of Computer Science, Iowa State University (August 2006) (to appear *Formal Aspects of Computing*)
16. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: *JML Reference Manual*, Draft revision 1.197 (August 2006)
17. Nanchen, S., Schmid, H., Schmitt, P., Stärk, R.F.: The ASMKeY prover. Technical Report 436, Department of Computer Science, ETH Zürich (2004)
18. van den Berg, J., Jacobs, B.: The loop compiler for java and jml. In: Margaria, T., Yi, W. (eds.) *ETAPS 2001 and TACAS 2001*. LNCS, vol. 2031, pp. 299–312. Springer, Heidelberg (2001)

Rebeca: Theory, Applications, and Tools

Marjan Sirjani^{1,2}

¹ University of Tehran, Tehran, Iran

² IPM School of Computer Science, Tehran, Iran

Abstract. Rebeca is an actor-based language with a formal foundation for modeling concurrent and distributed systems which is designed in an effort to bridge the gap between formal verification approaches and real applications. Rebeca is supported by a tool-set for model checking Rebeca models. Inherent characteristics of Rebeca are used to introduce compositional verification, abstraction, symmetry and partial order reduction techniques for reducing the state space. Simple message-driven object-based computational model, Java-like syntax, and set of verification tools make Rebeca an interesting and easy-to-learn model for practitioners. This paper is to present theories, applications, and supporting tools of Rebeca in a consistent and distilled form.

Keywords: Actors, Rebeca, Distributed Systems, Concurrency, Compositional Verification, Model Checking, Abstraction.

1 Introduction

In the last decades, computing technology has shifted from centralized mainframes to networked embedded and mobile devices, with the corresponding shift in applications from number crunching and data processing to the Internet and distributed computing. Concurrent and reactive systems are increasingly used in applications, and correct and highly dependable construction of such systems is particularly important and challenging. A very promising and increasingly attractive method for achieving this goal is using the approach of formal verification. *Reactive Objects Language, Rebeca* [1,2], is an actor-based language which is designed to bridge the gap between practical software engineering domains and formal verification methods.

Actors. Rebeca is an operational interpretation of *actor* model [3,4,5]. The actor model is a mathematical model of concurrent computation that treats *actors* as the universal primitives of concurrent computation: in response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received. Actors have encapsulated states and behavior; and are capable of creating new actors, and redirecting communication links through the exchange of actor identities. The actor model was originally introduced by Hewitt [3] as an agent-based language back in 1973 for programming secure distributed systems with multiple users and multiple security domains. It has been used both as a framework for a

theoretical understanding of concurrency, and as the theoretical basis for several practical implementations of concurrent and distributed systems.

Object-oriented models for concurrent systems have widely been proposed since the 1980s [5,6]. The actor model was developed by Agha [5,7,4] as a functional concurrent object-based model. Later several imperative languages have also been developed based on it [8,9,10]. Valuable work has been done on formalizing the actor model [4,11,12,13,14].

Rebeca modeling. Rebeca has a Java-like syntax and an operational formal semantics denoted by Labeled Transition Systems (LTS) [2]. Using Rebeca, systems are modeled as concurrently executing reactive objects. Reactive objects, called rebecs, are encapsulated objects with a set of state variables, known rebecs, and message servers. Known rebecs are those rebecs which messages are sent to. Communication takes place by asynchronous message passing which is non-blocking for both sender and receiver. The sender rebec sends a message to the receiver rebec and continues its work. There is no explicit receive statement. The message is put in the unbounded mail queue of the receiver and is serviced when it is taken from the queue by executing the corresponding message server. Execution of a message server takes place atomically. Although different kinds of synchronization patterns can be modeled using asynchronous computational model of Rebeca, extensions of Rebeca are developed which are equipped with statements to model synchronous patterns in a more straight forward way [15,16].

Formal verification. The current and future main challenges of formal verification is finding the ways to tackle state explosion in model checking by using: more efficient data structures and algorithms, composition and decomposition, abstraction, symmetry and partial order reduction. Other attempts include developing methods for verifying parameterized designs, develop practical tools for real-time and hybrid systems, and developing tool interfaces suitable for system designers [17]. Formal verification can be integrated with the system development process, from requirements analysis, to design and implementation (refinement process), and in testing. There is a strong trend to use model checking techniques for testing and debugging, or finding the performance, instead of proving the correctness of the system. Furthermore, verification techniques and approaches shall be provided in a more user-friendly form and be taught to students and software practitioners, at all levels, to make it more widely used.

Rebeca verification. Formal semantics of Rebeca is a solid basis for its formal verification. Compositional verification (decomposition), modular verification (composition), abstraction, symmetry and partial order reduction are all investigated for verifying Rebeca models, theories are established and tools are generated [18,19,16,20,21]. Rebeca with its simple message-driven object-based computational model, Java-like syntax, and set of verification tools is an interesting and easy-to-learn model for students and practitioners. Rebeca can be integrated in system development process by using Rebeca UML profile [22,23] to draw the diagrams for analysis and design modeling. Models can be refined, mapped to Rebeca, and model checked (and this process may iterate). The final

Rebeca code can be mapped to corresponding Java code [24]. Randomized model checking and performance evaluation using Rebeca are ongoing research.

Rebeca applications. Different computing paradigms may be used in modeling distributed and concurrent systems. The network setting can be assumed to be synchronous, asynchronous, or partially synchronous. Interprocess communication mechanism can be by shared memory or message passing [25]. Modeling in an asynchronous setting with message passing may be more difficult than modeling in other paradigms because of the more uncertainty involved. But, there are applications, like network protocols and web services, where this is the natural setting. While Rebeca can be used for modeling all patterns of concurrent systems, it can most effectively be used for modeling these kinds of systems [26,27,28].

Rebeca tools. Rebeca is supported by a number of formal verification tools which are now gathered in an integrated tool. There are tools to translate Rebeca models to the modeling language of back-end model checkers [19,29,18,30], SMV [31], Promela [32], and mCRL2 [33]. A direct model checker of Rebeca, Modere [21], in which symmetry and partial order reduction techniques are incorporated [20,34] is developed. There is a tool for drawing UML diagrams of Rebeca models [23] and then translate them to Rebeca and a tool for mapping Rebeca to Java [24].

Key features of Rebeca. In modeling a concurrent system using Rebeca, reactive objects which are building modules of the model are also the units of concurrency (a feature inherited from actors). This facilitates the process of modeling a distributed and concurrent system using Rebeca. The asynchronous communication between the reactive objects makes them to be loosely coupled modules. This feature, together with non-blocking and event-driven nature of computation make Rebeca suitable for modeling network applications. At the same time, these features are used to apply compositional verification and to establish reduction techniques in model checking Rebeca models. The experimental results show that significant reductions are gained in certain kinds of applications.

We extended Rebeca to support synchronous communication in order to cover a wider domain of applications in a more natural way. It is made clear where using this extended version breaks down the compositionality or applicability of reduction techniques in verification. According to our experience, the object-oriented paradigm in modeling and its Java-like syntax make Rebeca an easy-to-use language for practitioners (familiar with programming) comparing to algebraic or logic-based modeling languages and even visual and graphical languages.

This is an overview paper, with the goal of putting together all the work that has been done on Rebeca in a single and consistent piece. All the experience of working with Rebeca are distilled and divided into sections of theories, applications and tools. A set of simple examples are provided to show concepts of modeling and verification (not published before). For more complicated and real-world examples we refer to the published papers. Some of the results and discussions on modeling and verification can be applied to the other actor-based

models, or even more general, to the concurrent models in which computation is message-driven and asynchronous.

Layout of the paper. Section 2 describes related work. In Section 3, we first show how to model concurrent systems using Rebeca. Then, in Section 4 formal verification approaches for Rebeca models are discussed. Section 5 shows the application categories which can be efficiently modeled using Rebeca and Section 6 is a brief overview of the available tools for Rebeca. Section 7 explains the future work.

2 Related Work

Rebeca can be compared in different aspects with other modeling languages of concurrent and distributed systems, e.g., power of computational model, ease of use, compositionality, and formal basis (from the modeling point of view); and verifiability of models, applicability of reduction techniques in model checking, and availability of tools (from the verification point of view).

Examples of the modeling languages which provide powerful computational models and formal basis are: process algebraic languages like CSP [35] and CCS [36]; automata-based languages like I/O Automata [25], and process-based languages like RML [37]. CSP is supported by an analysis and theorem proving tool: FDR (Failures/Divergences Refinement) [38]. CCS models are usually verified by equivalency checking. An extension on bisimulation is used by Larsen and Milner in [39] for a compositional correctness proof of a protocol modeled by CCS. In [25] the authors showed how to construct modular and hierarchical correctness proofs for I/O Automata models. RML is supported by the model checker Mocha [40] where a subset of linear temporal logic, alternating-time temporal logic, is used to specify the properties [41]. RML supports compositional design and verification.

When the main concern is verification, sometimes subsets of programming languages are used in modeling and are then model checked. For example, the NASA's Java PathFinder [42] is a translator from a subset of Java to Promela [32]. Its purpose is to establish a framework for verification and debugging of Java programs based on model checking. The Bandera Tool Set [43] is an integrated tool which takes Java source code and the properties written in Bandera's temporal specification language as input, and it generates a program model and the properties in the input language of one of the existing back-end model-checkers.

Using modeling languages of widely used model checkers is another alternative for modeling problems. NuSMV [31] and Spin [32] are two widely used and successful model checking tools. NuSMV is a tool for checking finite state systems against specifications in the temporal logics LTL (Linear Temporal Logic) and CTL (Computation Tree Logic) [44]. Spin uses a high level process-based language for specifying systems, called Promela (Process meta language) and LTL is its property specifying language. These languages are designed for model checking purposes and their formal semantics are usually not explicitly given. Using these tools needs certain expertise.

Compositional verification and abstraction are ways to tackle state explosion problem. In compositional verification, we deduce the overall property specification of the system from local properties of its constituents [45,46,47]. Abstraction techniques are based on abstracting away some details from the model and proving the equivalence of the original model and the abstracted one [48,49,50]. A similar approach to ours in using abstraction techniques is used in [51] for model checking open SDL systems.

Rebeca inherited the simplicity and power of actor computational model. Having an imperative interpretation of actors, and a Java-based syntax make Rebeca easy to use for practitioners. Rebeca is not compositional at modeling level, but it is supported by a compositional verification theorem based on its formal semantics. Efficient reduction and abstraction techniques are applicable due to the object-oriented paradigm (no shared variables) and loosely coupled modules (event-driven computation, non-blocking and asynchronous communication) of Rebeca. Tools are available for model checking and modular verification of Rebeca models. Experimental results show that incorporating the reduction techniques in Rebeca direct model checker gives us significant reduction in state space in a vast domain of applications.

3 Modeling

In actor-based computational model of Rebeca, each rebec is not only an encapsulation of data and procedures but also the unit of concurrency. This unification of data abstraction and concurrency is in contrast to language models, such as Java, where an explicit and independent notion of thread is used to provide concurrency. By integrating objects and concurrency, actors free the programmer from handling mutual exclusion to prevent harmful concurrent access to data within an object [52].

In the following we will first show the syntax and semantics of Rebeca. Then, extending Rebeca by a synchronization structure is briefly explained. Further, we explain the concept of component in Rebeca and how (and if) rebecs can be coordinated exogenously. At the end, we show how Rebeca is integrated in software development life cycle.

3.1 Abstract Syntax

Figure 1 shows the BNF of Rebeca syntax. As shown in this figure, a Rebeca model consists of the definitions of *reactive classes* and then the *main* part which includes rebec instantiations. When defining a reactive class, queue length is specified. Although rebec queues are unbounded in theory, this is not possible for model checking, so the modeler defined a length for the queue. Reactive classes have three parts of *known rebecs*, *state variables*, and *message server* definitions. A rebec can send messages to its known rebecs. State variables build a part of the state. Each message server has a name, a (possibly empty) list of parameters, and the message server body which includes the statements. The statements may be assignments, sending messages, selections, and creating new rebecs.

Example 1 (A simple Rebeca model). A simple example is shown in Figure 2. There are two reactive classes of `Producer` and `Consumer`. Rebecs `producer` and `consumer` are instantiated from these reactive classes, respectively. The `producer` starts by executing its initial message server in which a message `produce` is sent to itself. By executing its `produce` message server, the `producer` generates a data and sends it as a parameter of the message `consume` to the `consumer`. Then it sends a `produce` message to itself which models iteration and keeps the `producer` to be executed infinitely. The `consumer` stays idle until it receives a message `consume` from the `producer`. The list of known rebecs of the `consumer` is empty as it does not send messages to any rebec. Producing a data is modeled as a nondeterministic assignment in the `produce` message server (`p=?(1,2,3,4)`). It means that `p` gets one of the values of one to four, nondeterministically. Nondeterministic assignments can also be used for simulating inputs of a model. There is no explicit buffer for storing the produced data in this example, the message queue of the `consumer` acts like a buffer. More complicated models of producer-consumer example with a buffer and a dynamic buffer are shown in 2.

3.2 Semantics

In 2, the formal semantics of Rebeca is expressed as a *labeled transition system (LTS)*. Here, we use a simplified version of the semantics from 34. A Rebeca model is constructed by parallel composition of a set of rebecs, written as $R = \parallel_{i \in I} r_i$, where I is the set of the indices used for identifying each rebec. The number of rebecs, and hence I , may change dynamically due to the possibility of dynamic rebec creation. Each rebec is instantiated from a reactive-class (denoting its type) and has a single thread of execution. A reactive-class defines a set of state variables that constitute the local state of its instances.

Rebecs communicate by sending asynchronous messages. Each rebec r_i has a static ordered list of known rebecs, whose indices are collected in K_i . Rebec r_i can send messages to rebec r_j when $j \in K_i$. Nevertheless, rebecs may have variables that range over rebec indices (called rebec variables). These variables can also be used to designate the intended receiver when sending a message. By changing the values of rebec variables, one can dynamically change the topology of a model. The static topology of a system can be represented by a directed graph, where nodes are rebecs, and there is an edge from r_i to r_j iff $j \in K_i$.

The messages that can be serviced by rebec r_i are denoted by the set M_i . In the reactive class denoting the type of r_i , there exists a message server corresponding to each element of M_i . There is at least a message server ‘*initial*’ in each reactive class, which is responsible for initialization tasks. Message servers are executed atomically; therefore, each message server corresponds to an action.

Each rebec has an unbounded queue for storing its incoming messages. A rebec is said to be enabled if its queue is not empty. In that case, the message at the head of the queue determines the enabled action of that rebec. The behavior of a Rebeca model is defined as the interleaving of the enabled actions of the enabled rebecs at each state. At the initialization state, a number of rebecs are created statically, and an ‘*initial*’ message is implicitly put in their queues. The

```

<rebeca_model> ::=
  (<reactiveclass_definition>)+ <main_definition>

<reactiveclass_definition> ::=
  "reactiveclass" <reactiveclass_name> "(" <queue_length> ")" <reactiveclass_body>

<reactiveclass_body> ::=
  "{"
    <knownrebecs_definition>
    <statevars_definition>
    (<msgsrv_definition>)+
  "}"

<knownrebecs_definition> ::=
  "knownrebecs" "{"
    (<knownrebec_type> <name> ";")*
  "}"

<statevars_definition> ::=
  "statevars" "{"
    (<statevar_type> <name> ";")*
  "}"

<msgsrv_definition> ::=
  "msgsrv" <message_name> "(" <parameters> ")" <message_server_body>

<message_server_body> ::=
  "{"
    <statements>
  "}"

<statements> ::=
  (<assignment>)* |
  (<send_message>)* |
  (<if_clause>)* |
  (<dynamic_creation>)*

<dynamic_creation> ::=
  <rebec_type> <rebec_instance> "="
  "new" <rebec_type> "(" <knownrebecs_binding> ")" ":" "(" <initial_message_parameters> ")" ";"

<main_definition> ::=
  "main" "{"
    (<main_rebec_instantiation>)*
  "}"

<main_rebec_instantiation> ::=
  <rebec_type> <name> "(" <knownrebecs_binding> ")" ":" "(" <initial_message_parameters> ")" ";"

```

Fig. 1. Rebeca Grammar (abstract)

execution of the model continues as rebecs send messages to each other and the corresponding enabled actions are executed.

Definition 1 (LTS). $R = \langle S, A, T, s_0 \rangle$ is called a labeled transition system, where:

- The set of global states is shown as S .
- s_0 is the initial state.
- A denotes the set of the actions (message servers) of different reactive classes.
- The transition relation is defined as $T \subseteq S \times A \times S$.

<pre> reactiveclass Producer(2) { knownrebecs { Consumer consumer; } statevars { byte p; } msgsrvv initial() { self.produce(); } msgsrvv produce() { // produce data p=?(1, 2, 3, 4); consumer.consume(p); self.produce(); } } </pre>	<pre> reactiveclass Consumer(2) { knownrebecs { } statevars { byte p; } msgsrvv initial() { } msgsrvv consume(byte data) { //consume data p = data; } } main { Producer producer(producer):(); Consumer consumer():(); } </pre>
--	---

Fig. 2. Rebeca model of a simple Producer-Consumer

Since instances of similar reactive classes have similar actions, actions of each rebec are indexed by the identifier of that rebec. In the case of a nondeterministic assignment in a message server, it is possible to have two or more transitions with the same action from a given state. We may write $s \xrightarrow{a_i} t$ for (s, a_i, t) .

Definition 2 (Data and Rebec Variables). *The variables for holding and manipulating data are called data variables. Rebec variables are those holding rebec indices. We may use a subscript ‘d’ to distinguish data variables and a subscript ‘r’ to distinguish rebec variables.*

Rebec variables can only be assigned to other rebec variables, compared for (in)equality, or used to specify the receiver of a send statement. They can also be assigned a dynamically created rebec.

Each rebec r_j has a message queue q , denoted as $r_j.q$ which holds the message id, sender id, and parameters of the message.

Definition 3 (Global State). *A global state is defined as the combination of the local states of all rebecs: $s = \prod_{j \in I} s_j$.*

Local state of a rebec r_j is the valuation of its variables and its queue.

Definition 4 (Initial State). *In the initial state s_0 , $r_j.q$ holds the initial message, the sender id is null, and parameters of the initial message are in the queue as well.*

Definition 5 (Transition Relation). *Transition relation $T \subseteq S \times A \times S$ is defined as follows. There is a transition $s \xrightarrow{a_j} t$ in the system, iff the action a from rebec r_j is enabled (i.e., $r_j.q_1 = a$) at state s , and its execution results in state t . The action a is executed atomically, and the sub-actions include message server removal and the statements in the corresponding message server. We define the different possible kinds of sub-actions that a transition $s \xrightarrow{a_i} t$ may contain.*

1. Message removal: *This sub-action includes the removal of the first element from the queue and implicitly exists in all the actions as the first sub-action.*
2. Assignment: *An assignment is a statement of the form ‘ $w = d$ ’, where w is a state variable in r_j . If w is a data variable, d represents an expression evaluated using the values of the data variables in state s . If w is a rebec variable, d should be a rebec variable. As a result of this assignment, the value of d is assigned to w in state t .*
3. Dynamic rebec creation: *This statement has the form ‘ $r_v = \text{new } rc(kr) : (p_1, \dots, p_z)$ ’, where r_v is the new rebec to be created, rc is the name of a reactive-class, kr represents the known rebecs of r , and the set of p_i show the parameters passed to the initial message. This sub-action results in a new index v being added to I (in state t), which is assigned to the newly created rebec. Hence, the global state t will also include the local state of r_v . In state t , the message initial is placed in $r_v.q$, and the sender of the message is denoted as the creator rebec.*
4. Send: *In dynamic Rebeca models, messages can be sent both to known rebecs and to rebec variables. Rebec r_j may send a message m with parameters n_1, \dots, n_z to r_k (the send statement $r_k.m(n_1, \dots, n_z)$), where $m \in M_k$, and either k belongs to K_j (known rebecs of r_j) or r_j has a rebec variable that holds the value k . In addition, n_i may be a data parameter or a rebec parameter. This send statement, results in the message m being placed in the tail of the queue of the receiving rebec.*

Example 2 (A simple Rebeca model with dynamic behavior). Figure 3 shows an alternative model for Producer-Consumer of Example 1 where the `producer` shows a dynamic behavior. If `p` gets the value three in the nondeterministic assignment, then a new `producer` is created. This can be interpreted as if three means that more producers are needed. If `p` gets the value four, then a the message `produce` will not be sent to itself, as a result of which the execution of the `producer` is terminated.

3.3 Synchronous Communication (Rendezvous)

In [15] and [16] the modeling power of the asynchronous and message-driven computational model of Rebeca is enriched by introducing a rendezvous-like synchronization between rebecs. Synchronous messages are specified only in terms of their signature: message name and parameters; they do not specify a corresponding body. Synchronous message passing involves a *hand-shake* between the execution of a send statement by the caller and a receive statement by the callee in which the (synchronous) message name specified by the caller is included. A *receive* statement, say *receive*(m_1, \dots, m_n), denotes a nondeterministic choice between receiving messages m_1 to m_n . This kind of synchronous message passing is a two-way blocking, one-way addressing, and one-way data passing communication. It means that both sender and receiver should wait at the rendezvous point, only sender specifies the name of the receiver, and data is passed from sender to receiver.

<pre> reactiveclass Producer(2) { knownrebecs { Consumer consumer; } statevars { byte p; Producer newProducer; } msgsrvv initial() { self.produce(); } msgsrvv produce() { // produce data p=?{1, 2, 3, 4}; if (p == 3) { newProducer = new Producer(consumer):(); } consumer.consume(p); if (p != 4) { self.produce(); } } } </pre>	<pre> reactiveclass Consumer(2) { knownrebecs { } statevars { byte p; } msgsrvv initial() { } msgsrvv consume(byte data) { //consume data p = data; } } main { Producer producer(consumer):(); Consumer consumer():(); } </pre>
--	---

Fig. 3. Rebeca model of Producer-Consumer with dynamic behavior

For adding synchronous messages to Rebeca we have the following changes in the grammar presented in Figure 1: a receive statement is added to the $\langle \text{statements} \rangle$ definition, and the body part in $\langle \text{msgsrvv-definition} \rangle$ becomes optional.

Example 3 (A Rebeca model with synchronous communication). Figure 4 shows a Rebeca code with a synchronous message. Here, we add a qualityController to the Producer-Consumer example (Example 1). The producer sends a message to the qualityController to control the quality of the product before passing it to the consumer. The producer waits until it receives the done synchronous message from qualityController. This pattern could be modeled without a synchronous message by adding extra asynchronous messages to encounter the necessary synchronization, but in a more complicated and less natural way.

3.4 Encapsulating Rebecs in Components

In [15] and [16] a component-based version of Rebeca is proposed where the components act as wrappers that provide higher level of abstraction and encapsulation. The main problem in constructing a component in this object-based configuration is the rebec-to-rebec communication and the need to know the receiver names. In [15] and [16] components are sets of rebecs and the communication between components is via broadcasting anonymous and asynchronous messages.

Send statements, where the receiver is mentioned are targeted to rebecs inside the component, while an anonymous *send* statement, which does not indicate

<pre> reactiveclass Producer(2) { knownrebecs { Consumer consumer; } statevars { byte p; } msgsrv initial() { self.produce(); } msgsrv produce() { // produce data p=?{1, 2, 3, 4}; qualityController.controlQuality(p); receive(done); consumer.consume(p); if (p != 4) { self.produce(); } } msgsrv done(); } reactiveclass QualityController(2) { knownrebecs { Producer producer; } statevars { } } </pre>	<pre> msgsrv initial() { } msgsrv controlQuality(p) { //control quality of p producer.done(); } } reactiveclass Consumer(2) { knownrebecs { } statevars { byte p; } msgsrv initial() { } msgsrv consume(byte data) { //consume data p = data; } } main { Producer producer(consumer):(); QualityController qualityController(producer):(); Consumer consumer():(); } </pre>
---	---

Fig. 4. Rebeca model of Producer-Consumer with synchronous message

the name of the receiver, causes an asynchronous broadcast of the message. This broadcast in fact will involve all the components of the system. Within a component, this in turn, will cause sending an asynchronous message to one of its rebecs which provides the service (non-deterministically chosen). Alternatively, it could also be broadcasted internally to all the rebecs of the receiving component (but the latter model is not consistent with the queue abstraction theory in formal verification approach, explain later in Section 4).

This setting allows us to put more tightly-coupled rebecs in a set and wrap them within a component. The rebecs within a component know each other and may also have some kind of synchronous communication. Hence, the compositional verification and abstraction techniques can still be applied where instead of reactive objects we take components as modules. This provides a setting capable of modeling globally asynchronous and locally synchronous systems which is supported by a modular verification approach. The modular verification theorem helps us to have verified components to be used (and reused) in design process (explain later in Section 4).

Example 4 (Components in a Rebeca model). In Example 3, we may put the `producer` and the `qualityController` in a component. The syntax is adding a declaration in the `main` part of the model which shows the set of rebecs which are included in a component (shown in Figure 5). For this specific example there is no messages provided to outside by the component including the `producer` and the `qualityController`. The only provided message of `producer` is `produce` which is

only sent to itself. The provided message of the `qualityController` is `controlQuality` which is sent to it by the `producer` and hence, it is not provided for serving outside rebecs. This example can be slightly changed such that for sending out a product, a request shall be first sent to the `producer` (then this request will be the provided message of the `producer`).

```

components:
  {producer, qualityController};
  {consumer};

```

Fig. 5. Adding components to Example 3, Figure 4

3.5 Coordinating Rebecs

Another way of defining components in Rebeca is discussed in [53]. In that work the possibility of mapping Rebeca models into a coordination language, Reo [54], is investigated and a natural mapping is presented. Reactive objects in Rebeca are considered as black-box components in Reo circuits while their behavior are specified using constraint automata [55]. Coordination and communication among reactive objects are modeled as Reo connectors.

Reo [54] is an exogenous coordination model wherein complex coordinators, called connectors are compositionally built out of simpler ones. The atomic connectors are a set of user-defined point-to-point *channels*. Reo can be used as a *glue language* for construction of connectors that orchestrate component instances in a component based system. Reo connectors impose different patterns of coordination on the components they connect. Constraint automata provide a compositional semantics for Reo. Behavior of components and Reo connectors can be both expressed by constraint automata, hence, the behavior of the whole circuit can be compositionally built up.

This work provides a compositional semantics for Rebeca, while the LTS semantics of Rebeca [2] is not compositional. Also, by re-directing or delaying delivery of messages, rebecs can be exogenously coordinated by the Reo connector. Although, the inherent behaviour of rebecs/actors which is non-blocking execution, cannot be changed.

3.6 From Specification to Implementation

The focus in designing Rebeca is on practice as well as theory. To build up a complete life-cycle support for developing systems, a UML profile is designed for Rebeca models [22]. The developer can start from UML diagrams using this profile, and then move to Rebeca code. The Rebeca code can be verified and possible necessary modifications are done. To facilitate moving from specification to implementation, a tool for mapping Rebeca codes to Java programs is

developed [24], making it possible to deploy Rebeca models on a real platform. Non-deterministic assignments mostly models the inputs to the model or some abstracted operation and shall be resolved for the translation. For that we need the interference of the modeler. Figure 6 shows the process of moving from specification to implementation.

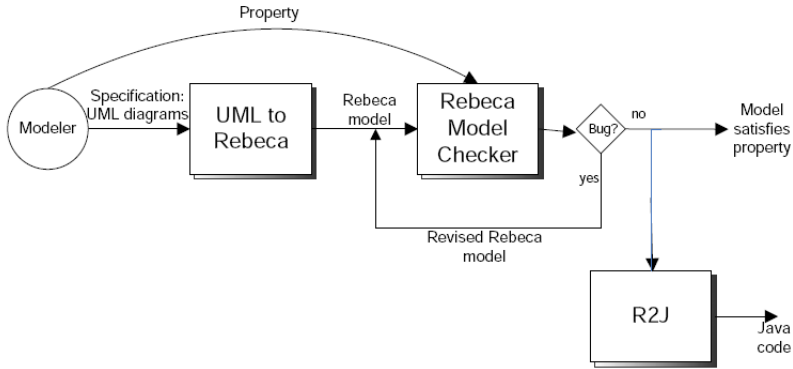


Fig. 6. From UML specification to Rebeca to Java

4 Formal Verification

Two basic approaches in formal verification are theorem proving and model checking. In theorem proving both the system and its desired properties are expressed in some mathematical logic and the goal is to find a proof for satisfaction of properties of the system. Model checking is exploring the state space of the system and checking whether the desired properties are satisfied. The main problem with theorem proving is formulating a complicated real system in a mathematical model and also the high expertise needed in proof process. The problem with model checking is state explosion.

A combination of model checking and deduction can be exploited in a number of ways, for example in abstraction and compositional verification. Compositional verification, abstraction, and reduction techniques are applied for Rebeca models [11,56,2,20,34]. A number of examples can be found on Rebeca Home page [24] and published papers.

4.1 Model Checking

Rebeca models can be verified by the supporting model checkers. We use temporal logic as our property specification language. The properties should be defined with temporal logic formulas. These formulas are based on state variables of the rebecs. The message queue contents are not considered in the properties.

Using Spin as the back-end model checker we can specify our properties as LTL-X (Linear Temporal Logic without next) formulas, and using Nu-SMV we

can specify our properties as both LTL and CTL (Computational Tree Logic) formulas. Sarir [30] makes it possible to have properties in μ -calculus. The direct model checker, Modere, handles properties in LTL. A CTL direct model checker for Rebeca is under development.

Example 5 (Verifying properties of a Rebeca model). In Example 2, Figure 3, we may want to check the following properties where G denotes *Globally* and F denotes *Finally* in LTL:

The variable p of the `consumer` rebec never gets the values of 3 or 4.

$$G(\neg(\text{consumer.p} = 3) \wedge \neg(\text{consumer.p} = 4))$$

The variable p of the `consumer` rebec will finally have values of 1 or 2.

$$F((\text{consumer.p} = 1) \vee (\text{consumer.p} = 2))$$

Abstraction techniques in model checking Rebeca models. Finite-state model-checkers are not able to deal with infinite state space, which is present in Rebeca due to the unboundedness of the queues. Thus, we need to impose an abstraction mechanism on our models: each rebec has a user-specified, finite upper bound on the size of its queue. The queue is checked for overflow condition during the model checking process, and in the case of overflow the queue length can be increased by the modeler. There are also conventional data abstractions which are regular for model checkers.

Another technique which is applied due to the special features of computation model of Rebeca is atomic method execution. This technique builds a course grained interleaving and causes a significant reduction in state space.

4.2 Compositional Verification

In *compositional verification*, the global property of a system is decomposed into the properties of its components which are then verified separately. If we deduce that the system satisfies each local property, and show that the conjunction of the local properties implies the global property, then we can conclude that the system satisfies this specification [45,46,47]. *Modular verification* [57,58] is usually used as a slightly different concept. In compositional verification we decompose a model for verification purposes where the complete model is initially designed. In modular verification, notion of component is used as a re-usable off-the-shelf module, once verified it has a fixed proven specification which can be used to build a reliable system in a bottom-up approach. The basic theories of these two approaches are similar because they are both for verifying open systems. The notion of environment and the interface between components and the environment shall be considered carefully and the technical details (to keep the theories valid) may be quite different. *Assume-guarantee* reasoning can also be applied where for model checking components you first assume some properties about the environment and then guarantee the desired properties for the components. In networks of finite state processes *induction* may be used to generalize the results on desired properties.

In [2], components are sub-models which are the result of decomposing a closed model in order to apply compositional verification. In decomposition, there is no general approach in deciding about the constituent rebecs of a component. Components have to be selected carefully to lead to a smaller state space and to help in proving the desired global properties. In [16], modular verification is discussed and the concept of a component is what we have in component-based modeling which is an independent module with a well-defined interface. In both cases we have a set of rebecs which make a component. This component is an open system reacting with an environment. The behavior of the internal rebecs of a component is fully modeled, except for the messages targeted to a rebec not inside the component (external rebecs, i.e., environment). The behavior of the environment shall be modeled as non-deterministically sending all possible messages to the internal rebecs.

Example 6 (Compositional verification of Rebeca models). As an example for a desired property of a component, we may consider the component including `producer` and `qualityController` in Example 3 (Figure 4). For this component, we want to assure that a product is not sent out (to the `consumer`) before quality control is done. As the properties shall be defined based on the state variables we may need to add some variables to show the rebec states.

4.3 Abstractions and Weak Simulation Relation

A way for state space reduction is to obtain an abstraction of a model, and then model check it. Equivalency relations (simulation/weak simulation) and accordingly property preservation shall be proved for the model and its abstraction (using deduction). Then the satisfaction of the desired property can be checked on the abstract model and claimed to be true for the model itself.

To model an environment which simulates all the possible behaviors of a real environment for Rebeca models, we need to consider an environment nondeterministically sending unbounded number of messages. We proved in [16.2] that with this arbitrary environment we are over-approximating the behavior of the component and there is a weak simulation relation between any closed model including the component and the component composed by the above mentioned environment.

It is clear that model checking will be impossible in this case where an environment is filling up the queues of the rebecs of the component. To overcome this problem, we use an abstraction technique. Instead of putting incoming messages in the queues of rebecs, they may be assumed as a constant (although unbounded) set of requests to be processed at any time, in a fair interleaving with the processing of the requests in the queue. This way of modeling the environment, generates a closed model which simulates the model resulting from a general environment which nondeterministically sends unbounded number of messages.

To formalize the discussion, consider a Rebeca model M , and a component C , where we denote the environment of C as E_C . If $R(M)$ be the transition system

of $M = C, E_C$ and $R(C_a)$ be the transition system generated by the transition relation considering queue abstraction, and $R(M')$ be the transition system of any model M' containing C , we have the following weak simulation relations: $R(C_a)$ weakly simulates $R(M)$ (Queue Abstraction), and $R(M)$ weakly simulates $R(M')$ (Component Composition). We also have the property preservation theorem: safety properties (LTL-X, ACTL-X) are preserved due to weak simulation [46]. Hence, if we prove a property for C_a by model checking, it can be deduced that the property also holds for an arbitrary model M' containing C .

4.4 Symmetry and Partial Order Reduction

The actor-based message-driven and asynchronous nature of Rebeca leads to more reduction techniques in model checking. In [20] and [34] the application of partial order and symmetry reduction techniques to model checking dynamic Rebeca models are investigated. A polynomial-time solution for finding symmetry-based equivalence classes of states is found while the problem in general is known to be as hard as graph isomorphism.

The symmetry reduction technique [59,60] takes advantage of structural similarities in the state-space. Viewing an LTS as a graph, the intuitive idea is to find those sub-graphs with the same structure, and constructing only one of these sub-graphs during state exploration.

In a given Rebeca model, the rebecs which are instantiated from the same reactive class exhibit similar behaviors. Considering the static communication relation among the rebecs in a model, the symmetry among rebecs can be detected automatically using known rebecs lists. This technique, called inter-rebec symmetry, does not need any special symmetry-related input from the modeler (unlike, for example, [60] and [61]). We can see this kind of symmetry in models with recurrent objects in a ring topology, like in dining philosophers. The approach can be extended to cover intra-rebec symmetry with the introduction of a notion of scalar sets to (the syntax and semantics) of Rebeca. Modeler can use the new syntax, to exhibit the internal symmetry of rebecs. This helps us to find symmetry in a wider range of topologies, like in star topology, where the internal symmetry of some objects is the key to the symmetry in the whole system.

Partial order reduction is an efficient technique for reducing the state-space size when model checking concurrent systems for Linear Temporal Logic without next (LTL-X) [62,63]. In Rebeca, concurrency is modeled by interleaving of the enabled actions from different rebecs. It is, however, not always necessary to consider all the possible interleaved sequences of these actions. The partial order reduction technique suggests that at each state, the execution of some of the enabled actions can be postponed to a future state, while not affecting the satisfiability of the correctness property. Therefore, by avoiding the full interleaving of the enabled actions, some states are excluded from the exhaustive state exploration in model checking [62,63]. The coarse-grained interleaving approach in Rebeca causes considerable reductions when partial order reduction is applied.

An action is called safe if it does not affect the satisfiability of the desired property (invisible) and is independent from all other actions of other rebecs

(globally independent). Intuitively, the execution of a safe action at a given state leads to a state where all other enabled actions (from other rebecs) remain enabled. So, the execution of other enabled actions can be postponed to the future states; and therefore, at each state we can define a subset of the enabled actions to be explored. In Rebeca, all the `assignment` statements are globally independent (due to the no-shared-variable criteria) and they are invisible and hence safe, if the variable in the left-hand-side of the assignment is not present in the property. All the `send` statements are invisible as the properties do not include queues (and queue contents). Send statements are globally independent and hence safe, if the sender rebec be the only rebec which sends messages to the receiver. The safe actions of each rebec can be statically determined and the subset of the enabled actions to be explored at each state is specified.

The experimental results, show significant improvements in model size and model-checking time when the techniques are applied in combination or separately (using Modere).

Example 7 (Compositional verification, symmetry and partial order reduction of Rebeca models). A good typical example which shows both compositional verification benefits and symmetry application is dining philosophers. There are n philosophers at a round table. To the left of each philosopher there is a fork, but s/he needs two forks to eat. Of course only one philosopher can use a fork at a time. If the other philosopher wants it, s/he just has to wait until the fork is available again. We can model this example in Rebeca with two reactive classes standing for the philosophers and forks. Then, we may instantiate any number of philosophers and forks in the main part of the model. The model can then be model checked using supporting tools.

For compositional verification of dining philosopher model we may consider two philosophers and the fork in between them as a component. This component can be used to prove the safety property indicating that a fork can not be in the hands of two philosophers simultaneously (mutual exclusion). For the progress property we shall consider two forks and the philosopher in between, and prove that the philosopher will finally have the both forks and eat (no deadlock). To prove the latter property we need to assume that each philosopher who starts to eat finally finishes eating and free the forks. This is an assumption necessary for the environment (external philosophers) sending messages to the internal forks. It will be interpreted as: if the fork receives a request from an external philosopher, it will eventually receives a free. This assumption helps us to guarantee the progress (no deadlock) property. For no starvation property, subtleties shall be considered in the Rebeca model (discussed in [34]).

In this model each philosopher knows its left and right forks (the forks are in its known rebecs list and the philosopher sends messages to forks to acquire them). Each fork also knows its left and right philosopher (and sends back the proper reply to their requests). This is an example of ring topology. Intuitively, we can see that philosophers (and similarly forks) have symmetric behavior. Also, the components used for compositional verification have symmetric behavior. Symmetry reduction can be applied for model checking this model. Partial order

reduction depends on the concrete model and the safe actions shall be identified base on the mentioned criteria.

5 Applications

Rebeca can be effectively used for concurrent, distributed, and asynchronous applications. Network and security protocols, web-services and agent-based systems fit well in this group. Mobile computing can be also modeled using dynamic features of Rebeca. To naturally model other kinds of applications, like for hardware and system design, some kind of synchronization features may be needed.

5.1 Network and Security Protocols

The nodes in the networks have internal buffers and communicate via asynchronous messages. Network protocols can be modeled by Rebeca by mapping each node of the network to a rebec. For network protocols an intruder is modeled as a rebec to verify the possibility of an attack.

There is an STP (Spanning Tree Protocol) definition in the IEEE 802.1D standard, based on the algorithm that was proposed by Perlman. In [28], a formal proof of the STP algorithm is presented by showing that finally a single node is selected as the root of the tree and the loops are eliminated correctly. Modular verification and induction are used in this proof. In [27] CSMA/CD protocol is modeled and verified using Rebeca. Again, each node in the network is modeled as a rebec.

A multi-phase Mitnick attack is modeled and model checked in [26]. The model consists of four rebecs: a server, a client, the TCPagent and an attacker. Attacker sends arbitrary packets to each host in network iteratively. In Mitnick attack a combination of simple attacks are used to reach the goal. First, the attacker floods the client by SYN packet to prevent it to respond to the server. Then by spoofing client address, and predicting server TCP sequence number connects to the server and invokes its dangerous command. Unpublished works are done on modeling and verifying Needham Schroeder attack and the security sub-layer in IEEE 802.16.

In all these works state explosion may occur immediately even with a small number of rebecs. To prevent this, attackers may be modeled as intelligent attackers which do not send all kinds of messages. Other reduction techniques, such as compositional verification and abstraction, are used for each specific case.

5.2 Hardware and System-Level Designs

The power of the computational model of Rebeca for modeling hardware and system-level designs is investigated in [64] and [65]. We found it natural in higher levels of abstraction as expected, and applicable in lower levels, and surprisingly very close to the behavior of real hardware components.

Software systems typically have an asynchronous model of execution, while hardware is usually designed for synchronous execution. In the higher levels

of abstraction in system-level design we generally have asynchronous models. Modeling the systems using Rebeca in the higher levels of abstraction, like the functional specification, is a natural process. At the beginning, there were doubts on the modeling process for the hardware design.

Hardware components run concurrently and react to the changes in their input by changing the output. This behavior naturally fits into the concept of reactive objects. So, each hardware component, sequential or combinational, is mapped to a rebec. Each input of a components is mapped to a state variable and a message server in the corresponding rebec. The state variable contains the value of the input and the message server is responsible to model the reaction of the component to a change in the input. The outputs of a component are provided by sending appropriate messages to the driven components (if an output of component A is an input of component B , the component A is called the driver and component B is called the driven component). In the rebecs which corresponds to sequential components, there is an additional message server reacting to the clock edge and sends messages according to its outputs. In the rebecs which corresponds to combinational components, in each message server related to the inputs, after setting the corresponding state variable the logic of the circuit is coded, and then the messages for outputs are sent. The data path of a design is mapped to the main part of the Rebeca model which includes the rebec instantiations and known rebecs relations. In order to model sequential circuits in which hardware components are synchronized by edges of a global clock signal, we need to introduce two additional rebecs in our model to perform the synchronization and clocking of the synchronous devices.

6 Tools

There is an integrated tool-set for model checking Rebeca models and a tool-set for integrating Rebeca in software development life cycle.

6.1 Model Checker Tools

The set of model checkers tool for Rebeca is shown in Figure 7. The tools do not yet support dynamic behaviors of Rebeca models.

Direct Model Checker: Modere. The direct model checker for Rebeca [20,21], is an LTL (Linear Temporal Logic) model checker which consists of two components: a translator and an engine. The Model-checking Engine of Rebeca (Modere) is the component that performs the actual task of model checking. It is based on the automata theoretic approach. In this approach, the system and the specification (for the negation of the desired property) are specified each with a Büchi automaton. The system satisfies the property when the language of the automaton generated by the synchronous product of these two automata is empty. Otherwise, the product automaton has an accepting cycle (a cycle containing at least one accepting state) that shows the undesired behavior of the system. In this case, an error trace witnessing a counter-example to the property is reported

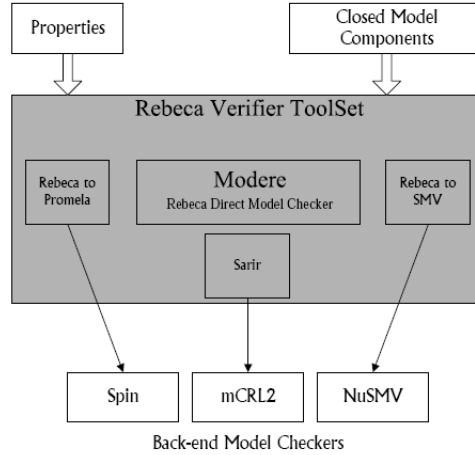


Fig. 7. Rebeca Model Checker tools

and the modeler can change the system model until the undesired behavior is eliminated. Given a Rebeca model and some LTL specification (for the negation of the desired property), the translator component generates the automata for the system and the specification. These automata are represented by C++ classes. The files containing these classes are placed automatically beside the engine (Modere). The whole package is then compiled to produce an executable for model checking the given Rebeca model. Modere, applies symmetry and partial order reduction techniques, as well as some algorithmic tricks to reduce memory usage.

R2SMV and R2Promela. Before implementing Modere, translators have been developed [29,19,18] to map Rebeca models to the language of back-end model checkers NuSMV [31] and Spin [32]. These tools, specially Rebeca to Promela, are still being used as Modere is still passing its final test phase and compositional verification is not yet incorporated in it. The property specification language of Spin is LTL-X (LTL without next operator) and NuSMV supports both LTL and CTL (Computational Tree Logic).

Rebeca to mCRL2: Sarir. mCRL2 [33] is a process algebraic language and an important goal in its design is to provide a common language for model checking (CRL stands for Common Representation Language). The main motivation for translating Rebeca to mCRL2 is to exploit the verification tools and the rich theories developed for mCRL2. The reduction theories will be investigated further to see how they can be incorporated in Rebeca direct model checker. The mapping is applied to several case-studies including the tree identification phase of the IEEE 1394 [30]. The results of the experiment show that the minimization tools of mCRL2 can be effective for model checking Rebeca. The state space

generated by mCRL2 is given to another tool set which supports mu-calculus as its property language.

6.2 UML Profile for Rebeca and Rebeca to Java

A UML profile is defined as a subset of UML concepts which is adequate to define our reactive applications (domain) [22] and a tool is provided based on this profile [23]. Using this tool we can convert our models to Rebeca, and then Rebeca to Java converter can help us to generate Java code from Rebeca models. Consequently, there would be a path from UML model to executable code that supports verification via Rebeca. The static structure of the model is captured by class and object diagrams and dynamic behavior is shown by sequence diagrams. There is a sequence diagram for each message server which shows the sequence of actions invoked by receiving a message. In each sequence diagram we have one main rebec and its known rebecs (because they are the only possible message receivers). The sequence of actions initiates by receiving a message from the sender, and the rest of the diagram shows how the receiver rebec services this message. State charts are not used in the profile, as in an abstract view, all the rebecs start in 'idle' state and after receiving a message go to 'waiting' state waiting for their turn, and then go to 'running' state.

By Rebeca to Java tool, rebecs are mapped to distributed objects and become independent Java processes. The distributed objects simulate the passing of messages in Rebeca by invoking the remote methods in Java (RMI). The Java process creates a thread for executing each message server. In the case that the message server contains a send statement, sending is carried out by making a new thread to execute the RMI.

7 Future Work

The future work is mostly focused on considering dynamic features in verification tools, investigating for more reduction techniques, and applying Rebeca to more real case studies in different application areas.

Acknowledgement

I wish to thank Mohammad Mahdi Jaghoori, Amin Shali, Hossein Hojjat, and Niloofar Razavi for reading the paper carefully and for their useful comments.

References

1. Sirjani, M., Movaghar, A.: An actor-based model for formal modelling of reactive systems: Rebeca. Technical Report CS-TR-80-01, Tehran, Iran (2001)
2. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.: Modeling and verification of reactive systems using Rebeca. *Fundamenta Informatica* 63, 385–410 (2004)

3. Hewitt, C.: Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. In: MIT Artificial Intelligence Technical Report 258, Department of Computer Science, MIT, Cambridge (1972)
4. Agha, G., Mason, I., Smith, S., Talcott, C.: A foundation for actor computation. *Journal of Functional Programming* 7, 1–72 (1997)
5. Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA (1990)
6. America, P., de Bakker, J., Kok, J., Rutten, J.: Denotational semantics of a parallel object-oriented language. *Information and Computation* 83, 152–205 (1989)
7. Agha, G.: The structure and semantics of actor languages. In: de Bakker, J.W., de Roever, W.P., Rozenberg, G. (eds.) *Foundations of Object-Oriented Languages*, pp. 1–59. Springer, Berlin (1990)
8. Ren, S., Agha, G.: RTsynchronizer: language support for real-time specifications in distributed systems. *ACM SIGPLAN Notices* 30, 50–59 (1995)
9. Yonezawa, A.: *ABCL: An Object-Oriented Concurrent System*. Series in Computer Systems. MIT Press, Cambridge (1990)
10. Varela, C., Agha, G.: Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices* 36, 20–34 (2001)
11. Mason, I.A., Talcott, C.L.: Actor languages: Their syntax, semantics, translation, and equivalence. *Theoretical Computer Science* 220, 409–467 (1999)
12. Talcott, C.: Composable semantic models for actor theories. *Higher-Order and Symbolic Computation* 11, 281–343 (1998)
13. Talcott, C.: Actor theories in rewriting logic. *Theoretical Computer Science* 285, 441–485 (2002)
14. Gaspari, M., Zavattaro, G.: An actor algebra for specifying distributed systems: The hurried philosophers case study. In: Agha, G.A., De Cindio, F., Rozenberg, G. (eds.) *Concurrent Object-Oriented Programming and Petri Nets*. LNCS, vol. 2001, pp. 216–246. Springer, Heidelberg (2001)
15. Sirjani, M., de Boer, F.S., Movaghar, A., Shali, A.: Extended rebeca: A component-based actor language with synchronous message passing. In: *Proceedings of Fifth International Conference on Application of Concurrency to System Design (ACSD'05)*, pp. 212–221. IEEE Computer Society, Los Alamitos (2005)
16. Sirjani, M., de Boer, F.S., Movaghar, A.: Modular verification of a component-based actor language. *Journal of Universal Computer Science* 11, 1695–1717 (2005)
17. Clarke, E.M.: The birth of model checking. In: Ball, T., Jones, R.B. (eds.) *CAV 2006*. LNCS, vol. 4144, Springer, Heidelberg (2006)
18. Sirjani, M., Shali, A., Jaghoori, M., Iravanchi, H., Movaghar, A.: A front-end tool for automated abstraction and modular verification of actor-based models. In: *Proceedings of Fourth International Conference on Application of Concurrency to System Design (ACSD'04)*, pp. 145–148. IEEE Computer Society, Los Alamitos (2004)
19. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.: Model checking, automated abstraction, and compositional verification of Rebeca models. *Journal of Universal Computer Science* 11, 1054–1082 (2005)
20. Jaghoori, M.M., Sirjani, M., Mousavi, M.R., Movaghar, A.: Efficient symmetry reduction for an actor-based model. In: Chakraborty, G. (ed.) *ICDCIT 2005*. LNCS, vol. 3816, pp. 494–507. Springer, Heidelberg (2005)
21. Jaghoori, M.M., Movaghar, A., Sirjani, M.: Modere: The model-checking engine of rebeca. In: *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC 2006)*, Software Verificatin Track, pp. 1810–1815 (2006)

22. Alavizadeh, F., Sirjani, M.: Using UML to develop verifiable reactive systems. In: Proceedings of International Workshop on the Applications of UML/MDA to Software Systems (at Software Engineering Research and Practice - SERP'06), pp. 554–561 (2005)
23. Alavizadeh, F., HashemiNekoo, A., Sirjani, M.: ReUML: A UML profile for modeling and verification of reactive systems. In: Proceedings of ICSEA'07 (2007)
24. (Rebeca homepage), <http://khorshid.ut.ac.ir/~rebeca/>
25. Lynch, N.: Distributed Algorithms. Morgan Kaufmann, San Francisco, CA (1996)
26. Shahriari, H.R., Makarem, M.S., Sirjani, M., Jalili, R., Movaghar, A.: Modeling and verification of complex network attacks using an actor-based language. In: Proceedings of 11th Annual Int. CSI Computer Conference, pp. 152–158 (2006)
27. Sirjani, M., SeyedRazi, H., Movaghar, A., Jaghouri, M.M., Forghanizadeh, S., Mojdeh, M.: Model checking csma/cd protocol using an actor-based language. WSEAS Transactions on Circuit and Systems 3, 1052–1057 (2004)
28. Hojjat, H., Nokhost, H., Sirjani, M.: Formal verification of the ieee 802.1d spanning tree protocol using extended rebeca. In: Proceedings of the First International Conference on Fundamentals of Software Engineering (FSEN'05). Electronic Notes in Theoretical Computer Science, vol. 159, pp. 139–159. Elsevier, Amsterdam (2006)
29. Sirjani, M., Movaghar, A., Irvanachi, H., Jaghoori, M., Shali, A.: Model checking Rebeca by SMV. In: Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'03), Southampton, UK, pp. 233–236 (2003)
30. Hojjat, H., Sirjani, M., Mousavi, M.R., Groote, J.F.: Sarir: A rebeca to mCRL2 translator, accepted for ACS07 (2007)
31. (NuSMV), <http://nusmv.irst.itc.it/NuSMV>
32. (Spin), Available through <http://netlib.bell-labs.com/netlib/spin>
33. Groote, J.F., Mathijssen, A., van Weerdenburg, M., Usenko, Y.: From μ CRL to mCRL2: motivation and outline. In: Workshop of Essays on Algebraic Process Calculi (2006). Electronic Notes in Theoretical Computer Science, pp. 191–196. Elsevier, Amsterdam (2006)
34. Jaghoori, M.M., Sirjani, M., Mousavi, M.R., Movaghar, A.: Symmetry and partial order reduction techniques in model checking Rebeca (submitted to a journal)
35. Hoare, C.A.R.: Communications Sequential Processes. Prentice-Hall, Englewood Cliffs (NJ) (1985)
36. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes. Information and Computation 100, 1–77 (1992)
37. Alur, R., Henzinger, T.: Reactive Modules. Formal Methods in System Design: An International Journal 15, 7–48 (1999)
38. Roscoe, W.A.: Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1998)
39. Larsen, K.G., Milner, R.: A compositional protocol verification using relativized bisimulation. Information and Computation 99, 80–108 (1992)
40. Alur, R., Henzinger, T.A., Mang, F.Y.C., Qadeer, S.: MOCHA: Modularity in model checking. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 521–525. Springer, Heidelberg (1998)
41. Alur, R., Henzinger, T.: Computer aided verification. Technical Report Draft (1999)
42. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. International Journal on Software Tools for Technology Transfer 2, 366–381 (2000)

43. Dwyer, M., Hatcliff, J., Joehanes, R., Laubach, S., Pasareanu, C., Robby, V.W., Zheng, H.: Tool-supported program abstraction for finite-state verification. In: Proceedings of the 23rd International Conference on Software Engineering, pp. 177–187 (2001)
44. Emerson, E.A.: Temporal and Modal Logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 996–1072. Elsevier Science Publishers, Amsterdam (1990)
45. Lamport, L.: Composition: A way to make proofs harder. In: de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.) COMPOS 1997. LNCS, vol. 1536, pp. 402–407. Springer, Heidelberg (1998)
46. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts (1999)
47. McMillan, K.L.: A methodology for hardware verification using compositional model checking. *Science of Computer Programming* 37, 279–309 (2000)
48. Kesten, Y., Pnueli, A.: Modularization and abstraction: The keys to practical formal verification. In: Brim, L., Gruska, J., Zlatuška, J. (eds.) MFCS 1998. LNCS, vol. 1450, pp. 54–71. Springer, Heidelberg (1998)
49. Rushby, J.: Integrated formal verification: Using model checking with automated abstraction, invariant generation, and theorem proving. In: Dams, D.R., Gerth, R., Leue, S., Massink, M. (eds.) SPIN 1999. LNCS, vol. 1680. Springer, Heidelberg (1999)
50. Saidi, H., Shankar, N.: Abstract and model check while you prove. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 443–453. Springer, Heidelberg (1999)
51. Ioustinova, N., Sidorova, N., Steffen, M.: Closing open SDL-systems for model checking with DTSpin. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 531–548. Springer, Heidelberg (2002)
52. Agha, G., Thati, P., Ziaei, R.: Actors: A model for reasoning about open distributed systems. In: Formal methods for distributed processing: a survey of object-oriented approaches- Section: Dynamic reconfiguration, pp. 155–176. Cambridge University Press, Cambridge (2001)
53. Sirjani, M., Jaghoori, M.M., Baier, C., Arbab, F.: Compositional semantics of an actor-based language using constraint automata. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 281–297. Springer, Heidelberg (2006)
54. Arbab, F.: Reo: A channel-based coordination model for component composition. *Mathematical Structures in Computer Science* 14, 329–366 (2004)
55. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.: Modeling component connectors in reo by constraint automata. *Science of Computer Programming* 61, 75–113 (2006)
56. Sirjani, M., Movaghar, A., Mousavi, M.: Compositional verification of an object-based reactive system. In: Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS'01), Oxford, UK, pp. 114–118 (2001)
57. Kupferman, O., Vardi, M.Y., Wolper, P.: Module checking. *Information and Computation* 164, 322–344 (2001)
58. Vardi, M.Y.: Verification of open systems. In: Ramesh, S., Sivakumar, G. (eds.) Foundations of Software Technology and Theoretical Computer Science. LNCS, vol. 1346, pp. 250–267. Springer, Heidelberg (1997)
59. Emerson, E., Sistla, A.: Symmetry and model checking. *Formal Methods in System Design* 9, 105–131 (1996)
60. Ip, C., Dill, D.: Better verification through symmetry. *Formal methods in system design* 9, 41–75 (1996)

61. Sistla, A.P., Gyuris, V., Emerson, E.A.: Smc: a symmetry-based model checker for verification of safety and liveness properties. *ACM Transactions on Software Engineering Methodology* 9, 133–166 (2000)
62. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Clarke, E., Kurshan, R.P. (eds.) *CAV 1990. LNCS*, vol. 531, pp. 176–185. Springer, Heidelberg (1991)
63. Valmari, A.: A stubborn attack on state explosion. In: Clarke, E., Kurshan, R.P. (eds.) *CAV 1990. LNCS*, vol. 531, pp. 156–165. Springer, Heidelberg (1991)
64. Hakimipour, N., Razavi, N., Sirjani, M., Navabi, Z.: Modeling and formal verification of system-level designs. In: submitted to *FDL'07* (2007)
65. Kakoe, M.R., Shojaei, H., Sirjani, M., Navabi, Z.: A new approach for design and verification of transaction level models. In: *Proceedings of IEEE International Symposium on Circuit and Sytems (ISCAS 2007)*, IEEE Computer Society Press, Los Alamitos (to appear, 2007)

Learning Meets Verification

Martin Leucker

Institut für Informatik
TU München, Germany

Abstract. In this paper, we give an overview on some algorithms for learning automata. Starting with Biermann’s and Angluin’s algorithms, we describe some of the extensions catering for specialized or richer classes of automata. Furthermore, we survey their recent application to verification problems.

1 Introduction

Recently, several verification problems have been addressed by using learning techniques. Given a system to verify, typically its essential part is learned and represented as a regular system, on which the final verification is then carried out. The aim of this paper is to present these recent developments in a coherent fashion. It should serve as an annotated list of references as well as describe the main ideas of these approaches rather than to pin down every technical detail, as these can be found in the original literature.

From the wide spectrum of learning techniques, we focus here on learning automata, or, as it is sometimes called, on *inference* of automata. We then exemplify how these learning techniques yield new verification approaches, as has recently been documented in the literature. Note, by verification we restrict to model checking [21] and testing [16] techniques.

This paper consists of two more sections: In the next section, we introduce learning techniques for automata while in Section 3, we list some of their applications in verification procedures.

In Section 2, we first recall Biermann’s so-called *offline* approach and Angluin’s *online* approach to learning automata. Then, we discuss variations of the setup for online learning as well as describe domain specific optimizations. Furthermore, we sketch extensions to Angluin’s learning approach to so-called regular-representative systems, timed systems, and ω -regular languages. We conclude Section 2 by giving references to further extensions and implementations.

In Section 3, we show applications of learning techniques in the domain of verification. We start with the problem of minimizing automata, which follows the idea of learning a minimal automaton for a given system rather than to minimize the given system explicitly. *Black-box checking*, which renders black-box testing as learning and (white-box) model checking, is discussed next. We continue with the idea of learning assumptions in compositional verification. Next, we follow the idea that a (least) fixpoint of some functional might be learned rather than computed iteratively. Implicitly, this idea has been followed when

learning network invariants or learning representations for the set of reachable states in regular model checking [2]. We conclude Section 3 by referring to further applications.

2 Learning Algorithms for Regular Systems

The general goal of learning algorithms for *regular systems* is to identify a *machine*, usually of *minimal* size, that *conforms* with an *a priori fixed* set of strings or a (class of) machines.

Here, we take machines to be *deterministic finite automata* (DFAs) (over strings), though most approaches and results carry over directly to the setting of finite-state machines (Mealy-/Moore machines).

In general, two types of learning algorithms for DFAs can be distinguished, so-called *online* and *offline* algorithms. Offline algorithms get a fixed set of *positive* and *negative* examples, comprising strings that should be *accepted* and, respectively, strings that should be *rejected* by the automaton in question. The learning algorithm now has to provide a (minimal) automaton that accepts the positive examples and rejects the negative ones.

Gold [30] was among the first studying this problem. Typical offline algorithms are based on a characterization in terms of a constraint satisfaction problem (CSP) over the natural numbers due to Biermann [14].

A different approach was proposed in [54], though imposing stronger assumptions (prefix-closeness) on the set of examples. We also note that there are efficient algorithms inferring a *not necessarily minimal* DFA, like [44] or [48] (known as RPNI).

Online algorithms have the possibility to ask further *queries*, whether some string is in the language of the automaton to learn or not. In this way, an online algorithm can enlarge the set of examples in a way most suitable for the algorithm and limiting the number of minimal automata in question, i.e., the ones conforming to the set of examples so far.

A popular setup for an online approach is that of Angluin's L^* algorithm [6] in which a minimal DFA is learned based on so-called *membership* and *equivalence queries*. Using a pictorial language, we have a *learner* whose job is to come up with the automaton to learn, a *teacher* who may answer whether a given string is in the language as well an *oracle* answering whether the automaton \mathcal{H} currently proposed by the learner is correct or not. This setting is depicted in Figure 1(a).

Clearly, an online algorithm like Angluin's should perform better than offline algorithms like Biermann's. Indeed, Angluin's algorithm is polynomial while without the ability to ask further queries the problem of identifying a machine conforming to given examples is known to be NP-complete [31].

Note that there are slightly different approaches to query-based learning of regular languages based on *observation packs* or *discrimination trees*, which are compared to Angluin's approach in [8, 13].

In Angluin's setting, a teacher will answer queries either positively or negatively. In many application scenarios, however, parts of the machine to learn

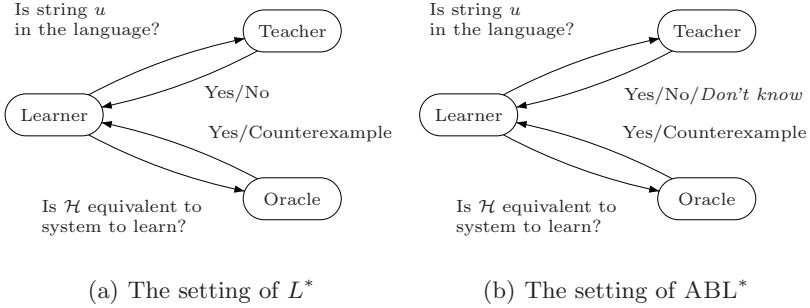


Fig. 1. The setup for the learning algorithms

are not completely specified or not observable. Then, queries may be answered inconclusively, by *don't know*, also denoted by ?. We term such a teacher *inexperienced*, see Figure 1(b).

In the following, we will look more closely at Biermann's approach (in Section 2.2), Angluin's algorithm (Section 2.3), as well as their combinations serving the setup with an inexperienced teacher (Section 2.4). Furthermore, we will sketch some extensions of these algorithms (Sections 2.5–2.9). Before, however, we look at the fundamental concept of right congruences that is the basis for the learning algorithms.

2.1 DFAs, Right-Congruences, and Learning Regular Systems

Let \mathbb{N} denote the natural numbers, and, for $n \in \mathbb{N}$, let $[n] := \{1, \dots, n\}$. For the rest of this section, we fix an alphabet Σ . A *deterministic finite automaton* (DFA) $\mathcal{A} = (Q, q_0, \delta, Q^+)$ over Σ consists of a finite set of *states* Q , an *initial state* $q_0 \in Q$, a *transition function* $\delta : Q \times \Sigma \rightarrow Q$, and a set $Q^+ \subseteq Q$ of *accepting states*. A *run* of \mathcal{A} is a sequence $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$ such that $a_i \in \Sigma$, $q_i \in Q$ and $\delta(q_{i-1}, a_i) = q_i$ for all $i \in [n]$. It is called *accepting* iff $q_n \in Q^+$. The *language* accepted by \mathcal{A} , denoted by $\mathcal{L}(\mathcal{A})$, is the set of strings $u \in \Sigma^*$ for which an accepting run exists. Since the automaton is deterministic, it is reasonable to call the states $Q \setminus Q^+$ also *rejecting states*, denoted by Q^- . We extend δ to strings as usual by $\delta(q, \epsilon) = q$ and $\delta(q, ua) = \delta(\delta(q, u), a)$. The size of \mathcal{A} , denoted by $|\mathcal{A}|$, is the number of its states Q , denoted by $|Q|$. A language is *regular* iff it is accepted by some DFA.

The basis for learning regular languages is given by their characterization in terms of *Nerode's right congruence* $\equiv_{\mathcal{L}}$: Let $\equiv_{\mathcal{L}}$ be defined by, for $u, v \in \Sigma^*$

$$u \equiv_{\mathcal{L}} v \text{ iff for all } w \in \Sigma^* : uw \in \mathcal{L} \Leftrightarrow vw \in \mathcal{L}.$$

It is folklore, that a language \mathcal{L} is regular iff $\equiv_{\mathcal{L}}$ has finite index.

Intuitively, most learning algorithms estimate the equivalence classes for a language to learn. Typically, it assumes that all words considered so far are equivalent, unless, a (perhaps empty) suffix shows that they cannot be equivalent.

Based on Nerode's right congruence, we get that, for every regular language \mathcal{L} , there is a *canonical* DFA $\mathcal{A}_{\mathcal{L}}$ that accepts \mathcal{L} and has a minimal number of states: Let $\mathbf{u}_{\mathcal{L}}$ or shortly \mathbf{u} denote the equivalence class of u wrt. $\equiv_{\mathcal{L}}$. Then the canonical automaton of \mathcal{L} is $\mathcal{A}_{\mathcal{L}} = (Q_{\mathcal{L}}, q_{0\mathcal{L}}, \delta_{\mathcal{L}}, Q_{\mathcal{L}}^+)$ defined by

- $Q_{\mathcal{L}} = \Sigma / \equiv_{\mathcal{L}}$ is the set of equivalence classes wrt. $\equiv_{\mathcal{L}}$,
- $q_{0\mathcal{L}} = \epsilon$,
- $\delta_{\mathcal{L}} : Q_{\mathcal{L}} \times \Sigma \rightarrow Q_{\mathcal{L}}$ is defined by $\delta_{\mathcal{L}}(\mathbf{u}, a) = \mathbf{ua}$,
- $Q_{\mathcal{L}}^+ = \{\mathbf{u} \mid u \in \mathcal{L}\}$

We omit the subscript \mathcal{L} provided \mathcal{L} is known from the context.

2.2 Biermann's Algorithm

Biermann's learning algorithm [14] is an offline algorithm for learning a DFA \mathcal{A} . We are given a set of strings that are to be accepted by \mathcal{A} and a set of strings that are to be rejected by \mathcal{A} . There is no possibility of asking queries and we have to supply a minimal DFA that accepts/rejects these strings. The set of positive and negative strings are called *sample*. We now formally describe samples and Biermann's algorithm.

A *sample* is a set of strings that, by the language in question, should either be accepted, denoted by $+$, or rejected, denoted by $-$. For technical reasons, it is convenient to work with prefix-closed samples. As the samples given to us are not necessarily prefix closed we introduce the value *maybe*, denoted by $?$. Formally, a *sample* is a partial function $O : \Sigma^* \rightarrow \{+, -, ?\}$ with finite, prefix-closed domain $\mathcal{D}(O)$. That is, $O(u)$ is defined only for finitely many $u \in \Sigma^*$ and is defined for $u \in \Sigma^*$ whenever it is defined for some ua , for $a \in \Sigma$. For a string u the sample O yields whether u should be *accepted*, *rejected*, or we do not know (or do not care). For strings u and u' , we say that O *disagrees* on u and u' if $O(u) \neq ?$, $O(u') \neq ?$, and $O(u) \neq O(u')$.

An automaton \mathcal{A} is said to *conform* with a sample O , if whenever O is defined for u we have $O(u) = +$ implies $u \in \mathcal{L}(\mathcal{A})$ and $O(u) = -$ implies $u \notin \mathcal{L}(\mathcal{A})$.

Given a sample O and a DFA \mathcal{A} that conforms with O , let S_u denote the state reached in \mathcal{A} when reading u . As long as we do not have \mathcal{A} , we can treat S_u as a variable ranging over states and derive constraints for the assignments of such a variable. More precisely, let $\text{CSP}(O)$ denote the set of equations

$$\begin{aligned} & \{S_u \neq S_{u'} \mid O \text{ disagrees on } u \text{ and } u'\} \quad (\text{C1}) \\ \cup & \{S_u = S_{u'} \Rightarrow S_{ua} = S_{u'a} \mid a \in \Sigma, ua, u'a \in \mathcal{D}(O)\} \quad (\text{C2}) \end{aligned}$$

Clearly, (C1) and (C2) reflect properties of Nerode's right congruence: (C1) tests u and u' on the empty suffix and (C2) guarantees right-congruence. Let the domain of $\mathcal{D}(\text{CSP}(O))$ comprise the set of variables S_u used in the constraints.

An *assignment* of $\text{CSP}(O)$ is mapping $\Gamma : \mathcal{D}(\text{CSP}(O)) \rightarrow \mathbb{N}$. An assignment Γ is called a *solution* of $\text{CSP}(O)$ if it fulfils the equations over the naturals, defined in the usual manner. The set $\text{CSP}(O)$ is *solvable* over $[N]$ iff there is a solution with range $[N]$. It is easy to see that every solution of the CSP problem over the natural numbers can be turned into an automaton conforming with O . We sum-up:

Lemma 1 (Learning as CSP, [14]). *For a sample O , a DFA with N states conforming to O exists iff $\text{CSP}(O)$ is solvable over $[N]$.*

Proof. Let $\mathcal{A} = (Q, q_0, \delta, Q^+)$ be a DFA conforming with O having N states. Without loss of generality, assume that $Q = [N]$. It is easy to see that assigning the value $\delta(q_0, u)$ to each $S_u \in \mathcal{D}(\text{CSP}(O))$ solves $\text{CSP}(O)$.

On the other hand, given a solution Γ of $\text{CSP}(O)$ with range $[N]$, define $\mathcal{A} = (Q, q_0, \delta, Q^+)$ by

- $Q = [N]$,
- $q_0 = S_\epsilon$,
- $\delta : Q \times \Sigma \rightarrow Q$ is any function satisfying $\delta(n, a) = n'$, if there is $S_u, S_{ua} \in \mathcal{D}(\text{CSP}(O))$ with $S_u = n$, $S_{ua} = n'$. This is well-defined because of (C2).
- $Q^+ \subseteq Q$ is any set satisfying, for $S_u \in \mathcal{D}(\text{CSP}(O))$ with $S_u = n$, $O(u) = +$ implies $n \in Q^+$, $O(u) = -$ implies $n \notin Q^+$. This is well-defined because of (C1).

Let us give three simple yet important remarks:

- Taking a different value for every S_u , trivially solves the CSP problem. Thus, a solution of $\text{CSP}(O)$ within range $[\mathcal{D}(\text{CSP}(O))]$ exists.
- Then, a solution with *minimum range* exists and yields a DFA with a minimal number of states.
- From the above proof, we see that we typically cannot expect to get a *unique* minimal automaton conforming with O —in contrast to Angluin’s L^* algorithm, as we will see.

Lemma 1 together with the remarks above gives a simple non-deterministic algorithm computing in polynomial time a DFA for a given observation, measured with respect to $N := |\mathcal{D}(\text{CSP}(O))|$, which is sketched Algorithm 1. Note that by results of Gold [31], the problem is NP complete and thus, the algorithm is optimal.

Algorithm 1. Pseudo code for Biermann’s Learning Algorithm

- 1 % Input O
 - 2 Guess $n \in [N]$, where $N := |\mathcal{D}(\text{CSP}(O))|$
 - 3 Guess assignment Γ of variables in $\mathcal{D}(\text{CSP}(O))$
 - 4 Verify that Γ satisfies (C1) and (C2)
 - 5 Construct the DFA as described in Lemma 1
-

Pruning the search space of the CSP problem. While the previous algorithm is of optimal worst case complexity, let us make a simple yet important observation to simplify the CSP problem, which often pays off in practice [35]. We call a bijection $\iota : [N] \rightarrow [N]$ a *renaming* and say that assignments Γ and Γ' are *equivalent modulo renaming* iff there is a renaming ι such that $\Gamma = \iota \circ \Gamma'$.

Since names (here, numbers) of states have no influence on the accepted language of an automaton, we get

Lemma 2 (Name irrelevance). *For a sample O , $\Gamma : \mathcal{D}(\text{CSP}(O)) \rightarrow [N]$ is a solution for $\text{CSP}(O)$ iff for every renaming $\iota : [N] \rightarrow [N]$, $\iota \circ \Gamma$ is a solution of $\text{CSP}(O)$.*

The previous lemma can be used to prune the search space for a solution: We can assign numbers to state variables, provided different numbers are used for different states.

Definition 1 (Obviously different). *S_u and $S_{u'}$ are said to be obviously different iff there is some $v \in \Sigma^*$ such that O disagrees on uv and $u'v$. Otherwise, we say that S_u and $S_{u'}$ look similar.*

A CSP problem with M obviously different variables needs at least M different states, which gives us together with Lemma 1:

Lemma 3 (Lower bound). *Let M be the number of obviously different variables. Then $\text{CSP}(O)$ is not solvable over all $[N]$ with $N < M$.*

Note that solvability over $[M]$ is not guaranteed, as can easily be seen.

As a solution to the constraint system produces an automaton and in view of Lemma 2, we can fix the values of obviously different variables.

Lemma 4 (Fix different values). *Let S_{u_1}, \dots, S_{u_M} be M obviously different variables. Then $\text{CSP}(O)$ is solvable iff $\text{CSP}(O) \cup \{S_{u_i} = i \mid i \in [M]\}$ is solvable.*

The simple observation stated in the previous lemma improves the solution of a corresponding SAT problem defined below significantly, as described in 35.

Solving the CSP problem. It remains to come up with a procedure solving the CSP problem presented above in a reasonable manner. An explicit solution is proposed in 47. In 35, however, an efficient encoding as a SAT problem has been given, for which one can rely on powerful SAT solvers. Actually, two different encodings have been proposed: *binary* and *unary*.

Let n be the number of strings in $\mathcal{D}(O)$ and N be the size of the automaton in question. Then $\text{CSP}(O)$ has $\mathcal{O}(n^2)$ constraints. Using the binary SAT encoding yields $\mathcal{O}(n^2 N \log N)$ clauses over $\mathcal{O}(n \log N)$ variables. Totally, the unary encoding has $\mathcal{O}(n^2 N^2)$ clauses with $\mathcal{O}(nN)$ variables (see 35 for details).

While the first is more compact for representing large numbers, it turns out that the unary encoding speeds-up solving the resulting SAT problem.

2.3 Angluin's Algorithm

Angluin's learning algorithm, called L^* 6, is designed for learning a regular language, $\mathcal{L} \subseteq \Sigma^*$, by constructing a minimal DFA \mathcal{A} such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}$. In this algorithm a *Learner*, who initially knows nothing about \mathcal{L} , is trying to learn \mathcal{L} by asking a *Teacher* and an *Oracle*, who know \mathcal{L} , respectively two kinds of queries (cf. Figure 1(a)):

- A *membership query* consists of asking whether a string $w \in \Sigma^*$ is in \mathcal{L} .

- An *equivalence query* consists of asking whether a hypothesized DFA \mathcal{H} is correct, i.e., whether $\mathcal{L}(\mathcal{H}) = \mathcal{L}$. The *Oracle* answers *yes* if \mathcal{H} is correct, or else supplies a counterexample w , either in $\mathcal{L} \setminus \mathcal{L}(\mathcal{H})$ or in $\mathcal{L}(\mathcal{H}) \setminus \mathcal{L}$.

The *Learner* maintains a prefix-closed set $U \subseteq \Sigma^*$ of prefixes, which are candidates for identifying states, and a suffix-closed set $V \subseteq \Sigma^*$ of suffixes, which are used to distinguish such states. The sets U and V are increased when needed during the algorithm. The *Learner* makes membership queries for all words in $(U \cup U\Sigma)V$, and organizes the results into a *table* T that maps each $u \in (U \cup U\Sigma)$ to a mapping $T(u) : V \mapsto \{+, -\}$ where $+$ represents accepted and $-$ not accepted. In [6], each function $T(u)$ is called a *row*. When T is

- *closed*, meaning that for each $u \in U$, $a \in \Sigma$ there is a $u' \in U$ such that $T(ua) = T(u')$, and
- *consistent*, meaning that $T(u) = T(u')$ implies $T(ua) = T(u'a)$,

the *Learner* constructs a hypothesized DFA $\mathcal{H} = (Q, q_0, \delta, Q^+)$, where

- $Q = \{T(u) \mid u \in U\}$ is the set of distinct rows,
- q_0 is the row $T(\lambda)$,
- δ is defined by $\delta(T(u), a) = T(ua)$, and
- $Q^+ = \{T(u) \mid u \in U, T(u)(\lambda) = +\}$

and submits \mathcal{H} as an equivalence query. If the answer is *yes*, the learning procedure is completed, otherwise the returned counterexample u is used to extend U by adding all prefixes of u to U , and subsequent membership queries are performed in order to make the new table closed and consistent producing a new hypothesized DFA, etc. The algorithm is sketched in Algorithm 2.

Complexity. It can easily be seen that the number of membership queries can be bounded by $O(kn^2m)$, where n is the number of states of the automaton to learn, k is the size of the alphabet, and m is the length of the longest counterexample. The rough idea is that for each entry in the table T a query is needed, and $O(knm)$ is the number of rows, n the number of columns. The latter is because at most n equivalence queries suffice. To see this, check that for any closed and consistent T , there is a single and therefore unique DFA conforming with T (as opposed to Biermann’s approach). Thus, equivalence queries are performed with automata of strictly increasing size.

2.4 Learning from Inexperienced Teachers

In the setting of an *inexperienced teacher* (cf. Figure 1(b)), queries are no longer answered by either *yes* or *no*, but also by *maybe*, denoted by $?$. We can easily come up with a learning algorithm in this setting, relying on Biermann’s approach: First, the learner proposes the automaton consisting of one state accepting every string.¹ Then, the Learner consults the oracle, which either classifies the automaton as the one we are looking for or returns a counter example c . In the latter

¹ Proposing the automaton that rejects every string is equally OK.

Algorithm 2. Pseudo code for Angluin's Learning Algorithm

```

1 Function Angluin()
2   initialize (U,V,T)
3   repeat
4     while not(isClosed((U, V, T)) or not(isConsistent((U, V, T)))
5       if not(isConsistent((U, V, T)) then
6         find  $a \in \Gamma$ ,  $v \in V$  and  $u, u' \in U$  such that
7            $T(u) = T(u')$  and  $T(ua)(v) \neq T(u'a)(v)$ 
8         add  $av$  to  $V$ 
9         for every  $u \in U \cup U\Gamma$ 
10          ask membership query for  $uav$ 
11        if not(isClosed((U, V, T)) then
12          find  $u \in U$ ,  $a \in \Gamma$  such that  $T(ua) \neq T(u')$  for all  $u' \in U$ 
13          move  $ua$  to  $U$ 
14          for every  $a' \in \Gamma$  and  $v \in V$ 
15            ask membership query for  $u'a'a'v$ 
16        construct hypothesized automaton  $\mathcal{H}$ 
17        do an equivalence query with hypothesis  $\mathcal{H}$ 
18        if the answer is a counterexample  $u$  then
19          add every prefix  $u'$  of  $u$  to  $U$ .
20          for every  $a \in \Gamma$ ,  $v \in V$  and prefix  $u'$  of  $u$ 
21            ask membership query for  $u'v$  and  $u'av$ .
22        until the answer is 'yes' to the hypothesis  $\mathcal{H}$ 
23      Output  $\mathcal{H}$ .

```

case, c as well as its prefixes are added to the sample O : c with $+/-$ as returned by the oracle and the prefixes with $?$ (unless O is already defined on the prefix), and Biermann's procedure is called to compute a minimal DFA consistent with O . Again the oracle is consulted and either the procedure stops or proceeds as before by adding the new counter example and its prefixes as before.

However, the procedure sketched above requires to create a hypothesis and to consult the oracle for every single observation and does not make use of membership queries at all.

Often, membership queries are "cheaper" than equivalence queries. Then, it might be worthwhile to "round off" the observation by consulting the teacher, as in Angluin's algorithm.

We list the necessary changes to Angluin's algorithm [35] yielding algorithm ABL*. We keep the idea of a table but now, for every $u \in (U \cup U\Sigma)$, we get a mapping $T(u) : V \rightarrow \{+, -, ?\}$. For $u, u' \in (U \cup U\Sigma)$, we say that rows $T(u)$ and $T(v)$ *look similar*, denoted by $T(u) \equiv T(u')$, iff, for all $v \in V$, $T(u)(v) \neq ?$ and $T(u')(v) \neq ?$ implies $T(u)(v) = T(u')(v)$. Otherwise, we say that $T(u)$ and $T(v)$ are *obviously different*. We call T

- *weakly closed* if for each $u \in U$, $a \in \Sigma$ there is a $u' \in U$ such that $T(ua) \equiv T(u')$, and
- *weakly consistent* if $T(u) \equiv T(u')$ implies $T(ua) \equiv T(u'a)$.

Angluin’s algorithm works as before, but using the weak notions of closed and consistent. However, extracting a DFA from a weakly closed and weakly consistent table is no longer straightforward. However, we can go back to Biermann’s approach now.

Clearly, Angluin’s table (including entries with $?$) can easily be translated to a sample, possibly by adding prefixes to $(U \cup U\Sigma)V$ with value $?$ to obtain a prefix-closed domain.

Catering for the optimizations listed for Biermann’s algorithm, given a table $T : (U \cup U\Sigma) \times V \rightarrow \{+, -, ?\}$, we can easily approximate obviously different states: For $u, u' \in (U \cup U\Sigma)$, states S_u and $S_{u'}$ are obviously different, if the rows $T(u)$ and $T(u')$ are obviously different.

Overall, in the setting of an inexperienced teacher, we use Biermann’s approach to derive a hypothesis of the automaton in question, but use the completion of Angluin’s observation table as a heuristic to round the sample by means of queries.

2.5 Domain Specific Optimizations for Angluin’s Algorithm

Angluin’s L^* algorithm works in the setting of arbitrary regular languages. If the class of regular languages is restricted, so-called *domain specific* optimizations may be applied to optimize the learning algorithm [38]. For example, if the language to learn is known to be *prefix closed*, a positive string ua implies u to be positive as well, preventing to consult the teacher for u [11].

Pictorially, such a setting can easily be described by adding an *assistant* between the learner and the teacher. Queries are sent to the assistant who only consults the teacher in case the information cannot be deduced from context information [13]. Assistants dealing with *independent actions*, *I/O systems*, and *symmetrical systems* have been proposed in [38].

2.6 Learning of Regular Representative Systems

Angluin’s L^* algorithm identifies a DFA accepting a regular language. Such a language, however, might *represent* a more complicated object. In [15], for example, L^* is used to infer *message-passing automata* (MPAs) accepting languages of *message sequence charts* (MSCs).

Let us work out a setup that allows to learn systems using a simple modification of Angluin’s L^* algorithm. A *representation system* is a triple $(\mathcal{E}, \mathcal{O}, \mathcal{L})$, where

- \mathcal{E} is a set of *elements*,
- \mathcal{O} is a set of *objects*,
- and $\mathcal{L} : \mathcal{O} \rightarrow 2^{\mathcal{E}}$ is a *language function* yielding for an object o the set of elements it represents.

The objects might be classified into equivalence classes of an equivalence relation $\sim_{\mathcal{L}} \subseteq \mathcal{O} \times \mathcal{O}$ by $\mathcal{L} : o \sim o'$ iff $\mathcal{L}(o) = \mathcal{L}(o')$. Intuitively, objects could be understood

as subsets of \mathcal{E} . However, like with words and automata, subsets of languages could be infinite while automata are a finite representation of such an infinite set.

A further example is where objects are MPAs, elements are MSCs, and MPA represent MSC languages. Then, two MPAs are considered to be equivalent if they recognize the same MSC language.

Our goal is now to *represent* objects (or rather their equivalence classes) by regular word languages, say over an alphabet Σ , to be able to use L^* . An almost trivial case is given when there is a bijection from regular word languages to \mathcal{O} . As we will see, a simple framework can also be obtained, when there is bijection of \mathcal{O} to a factor of a subset of regular languages.

Let \mathcal{D} be a subset of Σ^* . The motivation is that only regular word languages containing at most words from \mathcal{D} are considered and learned. Furthermore, let $\approx \subseteq \mathcal{D} \times \mathcal{D}$ be an equivalence relation. We say that $L \subseteq \mathcal{D}$ is *\approx -closed* (or, closed under \approx) if, for any $w, w' \in \mathcal{D}$ with $w \approx w'$, we have $w \in L$ iff $w' \in L$.

Naturally, \mathcal{D} and \approx determine the particular class $\mathfrak{R}_{\min\text{DFA}}(\Sigma, \mathcal{D}, \approx) := \{L \subseteq \mathcal{D} \mid L \text{ is regular and closed under } \approx\}$ of regular word languages over Σ (where any language is understood to be given by its minimal DFA). Suppose a language of this class $\mathfrak{R}_{\min\text{DFA}}(\Sigma, \mathcal{D}, \approx)$ can be learned in some sense that will be made precise. For learning elements of \mathcal{O} , we still need to derive an object from a language in $\mathfrak{R}_{\min\text{DFA}}(\Sigma, \mathcal{D}, \approx)$.

To this aim, we suppose a computable bijective mapping $obj : \mathfrak{R}_{\min\text{DFA}}(\Sigma, \mathcal{D}, \approx) \rightarrow [\mathcal{O}]_{\sim}$ (where $[o]_{\sim} = \{o' \in \mathcal{O} \mid o' \sim o\}$). A typical situation is depicted in Fig 2, where the larger ellipse is closed under \approx ($w \approx w'$), whereas the smaller circle is not, as it contains u but not u' .

As Angluin’s algorithm works within the class of arbitrary DFA over Σ , its *Learner* might propose DFAs whose languages are neither a subset of \mathcal{D} nor satisfy the closure properties for \approx . To rule out and fix such hypotheses, the language inclusion problem and the closure properties in question are required to be *constructively decidable*, meaning that they are decidable and if the property fails, a *reason* of its failure can be computed.

Let us be more precise and define what we understand by a *learning setup*:

Definition 2. Let $(\mathcal{E}, \mathcal{O}, \mathcal{L})$ be a representation system. A learning setup for $(\mathcal{E}, \mathcal{O}, \mathcal{L})$ is a quintuple $(\Sigma, \mathcal{D}, \approx, obj, elem)$ where

- Σ is an alphabet,
- $\mathcal{D} \subseteq \Sigma^*$ is the domain,
- $\approx \subseteq \mathcal{D} \times \mathcal{D}$ is an equivalence relation such that, for any $w \in \mathcal{D}$, $[w]_{\approx}$ is finite,
- $obj : \mathfrak{R}_{\min\text{DFA}}(\Sigma, \mathcal{D}, \approx) \rightarrow [\mathcal{O}]_{\sim}$ is a bijective effective mapping in the sense that, for $\mathcal{A} \in \mathfrak{R}_{\min\text{DFA}}(\Sigma, \mathcal{D}, \approx)$, a representative of $obj(\mathcal{A})$ can be computed.

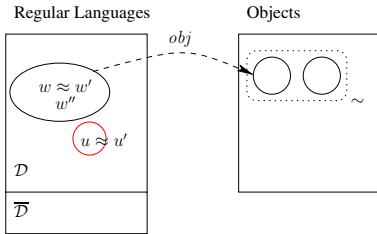


Fig. 2. Representing objects by regular languages

- $elem : [\mathcal{D}]_{\approx} \rightarrow \mathcal{E}$ is a bijective mapping such that, for any $o \in \mathcal{O}$,

$$elem(\mathcal{L}(obj^{-1}([o]_{\sim_{\mathcal{L}}})) = \mathcal{L}(o)$$

Furthermore, we require that the following hold for DFA \mathcal{A} over Σ :

- (D1) The problem whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{D}$ is decidable. If, moreover, $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{D}$, one can compute $w \in \mathcal{L}(\mathcal{A}) \setminus \mathcal{D}$. We then say that $\text{INCLUSION}(\Sigma, \mathcal{D})$ is constructively decidable.
- (D2) If $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{D}$, it is decidable whether $\mathcal{L}(\mathcal{A})$ is \approx -closed. If not, one can compute $w, w' \in \mathcal{D}$ such that $w \approx w'$, $w \in \mathcal{L}(\mathcal{A})$, and $w' \notin \mathcal{L}(\mathcal{A})$. We then say that the problem $\text{EQCLOSURE}(\Sigma, \mathcal{D}, \approx)$ is constructively decidable.

Given a regular representation system $(\mathcal{E}, \mathcal{O}, \mathcal{L})$ for a which a learning setup exists, let us sketch a learning algorithm that, by using membership queries for elements $e \in \mathcal{E}$ and equivalence queries for objects $o \in \mathcal{O}$, identifies a predetermined object. Let $(\Sigma, \mathcal{D}, \approx, obj, elem)$ be a learning setup for $(\mathcal{E}, \mathcal{O}, \mathcal{L})$. To obtain this algorithm, we rely on Angluin's algorithm but modify it a little. The general idea is that the algorithm learns the regular language representing the object in question. However, membership queries for words and equivalence queries for automata are translated thanks to $elem$ and respectively obj . However, use these functions in a meaningful manner, we modify also the processing of membership queries as well as the treatment of hypothesized DFAs:

- Once a membership query has been processed for a word $w \in \mathcal{D}$ (by querying $elem(w)$), queries $w' \in [w]_{\approx}$ must be answered equivalently. They are thus not forwarded to the *Teacher* anymore. Again, as in Section 2.5, we might think of an *Assistant* in between the *Learner* and the *Teacher* that checks if an equivalent query has already been performed. Membership queries for $w \notin \mathcal{D}$ are not forwarded to the *Teacher* either but answered negatively by the *Assistant*.
- When the table T is both closed and consistent, the hypothesized DFA \mathcal{H} is computed as usual. After this, we proceed as follows:
 1. If $\mathcal{L}(\mathcal{H}) \not\subseteq \mathcal{D}$, compute a word $w \in \mathcal{L}(\mathcal{H}) \setminus \mathcal{D}$, declare it a counterexample, and modify the table T accordingly (possibly involving further membership queries).
 2. If $\mathcal{L}(\mathcal{H}) \subseteq \mathcal{D}$ but $\mathcal{L}(\mathcal{H})$ is not \approx -closed, then compute $w, w' \in \mathcal{D}$ such that $w \approx w'$, $w \in \mathcal{L}(\mathcal{H})$, and $w' \notin \mathcal{L}(\mathcal{H})$; perform membership queries for $[w]_{\approx}$.

Actually, a hypothesized DFA \mathcal{H} undergoes an equivalence test (by querying $obj(\mathcal{H})$) only if $\mathcal{L}(\mathcal{H}) \subseteq \mathcal{D}$ and $\mathcal{L}(\mathcal{H})$ is \approx -closed. I.e., if, in the context of the extended learning algorithm, we speak of a hypothesized DFA, we actually act on the assumption that $\mathcal{L}(\mathcal{H})$ is the union of \approx -equivalence classes.

Let the extension of Angluin's algorithm wrt. a learning setup as sketched above be called EXTENDEDANGLUIN . A careful analysis shows:

Theorem 1. *Let $(\Sigma, \mathcal{D}, \approx, \text{obj}, \text{elem})$ be a learning setup for a representation system $(\mathcal{E}, \mathcal{O}, \mathcal{L})$. If $o \in \mathcal{O}$ has to be learned, then invoking*

$$\text{EXTENDEDANGLUIN}((\mathcal{E}, \mathcal{O}, \mathcal{L}), (\Sigma, \mathcal{D}, \approx, \text{obj}, \text{elem}))$$

returns, after finitely many steps, an object $o' \in \mathcal{O}$ such that $o' \sim_{\mathcal{L}} o$.

The theorem suggests the following definition:

Definition 3. *A representation system $(\mathcal{E}, \mathcal{O}, \mathcal{L})$ is learnable if there is some learning setup for $(\mathcal{E}, \mathcal{O}, \mathcal{L})$.*

2.7 Learning of Timed Systems

Angluin's algorithm has been extended to the setting of realtime systems. In [33] and [32], learning of *event-deterministic event-recording automata* and, respectively, *event-recording automata*, which both form sub-classes of timed automata [4], is described. For learning timed systems, several obstacles have to be overcome. First, timed strings range over pairs of letters (a, t) where a is from some finite alphabet while t is a real number, denoting the time when a has occurred. Thus, timed strings are sequences of letters taken from some *infinite alphabet*, while the learning algorithms deal with strings over finite alphabets. To be able to deal with strings over a finite alphabet, one joins several time points to get a *zone* [33] or *region* [32] of time points. This allows us to work over an alphabet consisting of actions and zone respectively region constraints, which, given a greatest time point K , gives rise to a finite alphabet. These strings, which are built-up from letters of actions and constraints, are sometimes also called *symbolic strings*. It has been shown that the set of symbolic strings accepted by an (event-deterministic) event-recording automaton forms a regular language [27], which we call the *symbolic regular language* of the automaton. The second obstacle to overcome is to derive a form or Nerode's right congruence for such symbolic regular languages that is consistent with a natural notion of right congruence for timed systems. This is implicitly carried out in [33][32] by introducing *sharply-guarded* event-deterministic event-recording automata and *simple* event-recording automata, which are both unique normal forms for all automata accepting the same language.

This gives that Angluin's learning algorithm can be reused to learn (symbolic versions of) timed languages, yielding either *sharply-guarded* event-deterministic event-recording automata [33] or *simple* event-recording automata [32]. However, a direct application of Angluin's algorithm employs queries for symbolic strings that might represent complex timed behavior of the underlying system. To deal with this obstacle, an assistant can then be used to effectively bridge the level from symbolic timed strings (actions plus regions) to timed strings (action plus time value).

A different approach to learning timed systems, based on decision trees, is presented in [34].

2.8 Learning of ω -Regular Languages

Reactive systems, like a web server, are conceptually non-terminating systems, whose behavior is best modelled by infinite rather than finite strings. In such a setting, Angluin's learning algorithm has to be extended to learn ω -regular languages [45]. Therefore, two main obstacles have to be overcome: First, a suitable representation for infinite strings has to be found. Second, a suitable version of Nerode's right congruence has to be defined. The first obstacle is overcome by considering only so-called ultimately periodic words, which can be described by $u(v)^\omega$ for finite words $u, v \in \Sigma^*$ [53]. The second obstacle is solved by restricting the class of ω -regular languages. The given algorithm is restricted to languages L for which both L and its complement can be accepted by a deterministic ω -automaton, respectively. This class coincides with the class of *deterministic weak Büchi automata*. We are not aware of any Biermann-style or Angluin-style learning algorithm for the full class of ω -regular languages.

2.9 Further Extensions

Especially in communication protocols, the input/output actions of a system can be distinguished in *control sensitive* or not. Some parameters of the system affect the protocol's state, while others are considered as *data*, just to be transmitted, for example. Optimizations of Angluin's algorithm for such a setup, in which we are given *parameterized actions*, have been proposed in [12].

Angluin's algorithm has been extended to deal with *regular tree languages* rather than word languages in [26,18]. *Learning of strategies for games* has been considered in [25]. A symbolic version of Angluin's L^* algorithm based on BDDs [17] is presented in [5].

Learning (certain classes of) message passing automata accepting (regular) sets of message sequence charts has been studied in [15], using ideas of regular representations (cf. Section 2.6).

2.10 Implementations

Implementations of Angluin's learning algorithm have been described and analyzed in [11,52].

3 Verification Using Learning

3.1 Minimizing Automata

Typical minimization algorithms for automata work on the transition graph of the given automaton. In [50], however, a minimization algorithm for incompletely specified finite-state machines was proposed, which is based on learning techniques: Instead of simplifying a given machine, a *new*, minimal machine is learned. For this, membership and equivalence queries are carried out on the given system.

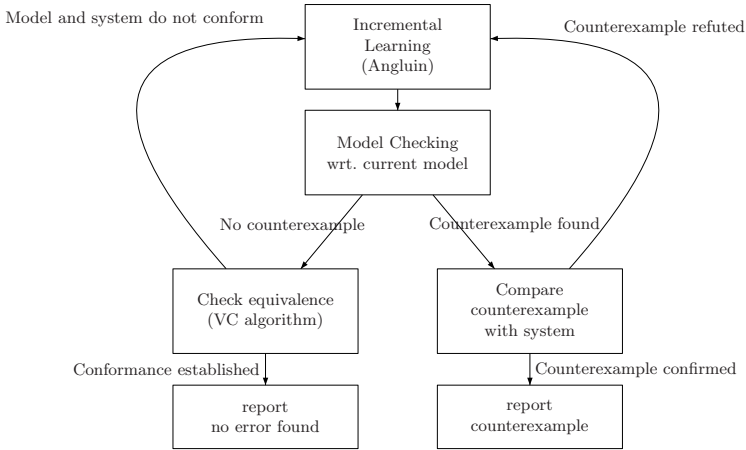


Fig. 3. Black Box Checking

Although no verification is performed in this approach, it entails one of the main motivations for using learning techniques in verification: Instead of deriving some object by modifying a large underlying system, one directly learns the small result. Clearly, it depends very much on the whole setup when this idea is fruitful.

3.2 Black-Box Checking

A different motivation for using learning techniques for verification purposes is when no model of the underlying system is given. Thus, we are faced with a so-called black box system i.e. a system for which no model is given but whose output can be observed for a given input. In *black box checking* or *adaptive model checking* [49,36] these systems should be tested against a formal specification. Conceptually, the problem can be solved by learning a (white box) model of the black box, on which model checking can be performed. This can be done—under strong restrictions—for example using L^* and a conformance test like the ones in [59,19]. In [49,36], the tasks of learning and model checking are interweaved as explained in Figure 3, suggesting better practical performance.

First, an initial model of the system to check is learned. If model checking stops with a counter example, this might be due to the inadequacy of the current version of the model. However, running the counter example reveals whether indeed a bug of the black-box system has been found, or, whether the counter example was spurious—and should be used to improve the model.

If model checking does not provide a counter example, we have to apply a conformance test [59,19] to make sure that no (violating) run of the black box is missing in the model. If a missing run was detected, the model is updated, otherwise, the correctness of the black box has been proved.

3.3 Compositional Verification

Compositional verification addresses the state-space explosion faced in model checking by exploiting the modular structure naturally present in system designs. One prominent technique uses the so-called *assume guarantee* rule: Take that we want to verify a property φ of the system $M_1 \parallel M_2$, consisting of two *modules* M_1 and M_2 running synchronously in parallel, denoted by $M_1 \parallel M_2 \models \varphi$. Instead of checking $M_1 \parallel M_2 \models \varphi$, one considers a module A and verifies that

1. $M_1 \parallel A \models \varphi$
2. M_2 is a refinement of A .

The rationale is that A might be simpler than M_2 , in the sense that both checking $M_1 \parallel A \models \varphi$ and M_2 is a refinement of A is easier than checking $M_1 \parallel M_2 \models \varphi$.

A setup, for which the assume-guarantee rule has been proven to be sound and complete [46], is when

1. modules can be represented as transition systems,
2. the parallel operator \parallel satisfies $\mathcal{L}(M \parallel M') = \mathcal{L}(M) \cap \mathcal{L}(M')$, and
3. φ is a safety property and can thus be understood as a DFA \mathcal{A}_φ accepting the allowed behavior of the system to check.

For such a setup, the assume-guarantee rule boils down to come up with some A such that

- (AG1) $\mathcal{L}(M_1 \parallel A) \subseteq \mathcal{L}(\mathcal{A}_\varphi)$ and
 (AG2) $\mathcal{L}(A) \supseteq \mathcal{L}(M_2)$.

In [24] and [5], it has been proposed to employ a learning algorithm to come up with such a module A . Here, we follow [5], which uses Angluin's L^* algorithm.

Whenever Angluin's algorithm proposes some hypothesis automaton A , it is easy to answer an equivalence query:

- if $\mathcal{L}(M_1 \parallel A) \not\subseteq \mathcal{L}(\mathcal{A}_\varphi)$, consider $w \in \mathcal{L}(M_1 \parallel A) \setminus \mathcal{L}(\mathcal{A}_\varphi)$. If $w \in \mathcal{L}(M_2)$, w witnesses that $M_1 \parallel M_2 \models \varphi$ does not hold. Otherwise, return w as a counterexample as result of the equivalence query.
- if $\mathcal{L}(A) \not\supseteq \mathcal{L}(M_2)$ provide a $w \in \mathcal{L}(A) \setminus \mathcal{L}(M_2)$ as a counterexample as result of the equivalence query.

If both tests succeed, we have found an A showing that $M_1 \parallel M_2 \models \varphi$ holds.

Membership queries are less obvious to handle. Clearly, when $w \in \mathcal{L}(M_2)$ then w must be in $\mathcal{L}(A)$ because of (AG2). If $w \notin \mathcal{L}(\mathcal{A}_\varphi)$ but $w \in \mathcal{L}(M_1)$, w must not be in $\mathcal{L}(A)$ since otherwise the safety property φ is not met ((AG1)). Note, if in this case also $w \in \mathcal{L}(M_2)$ (i.e., $w \in \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$, $w \notin \mathcal{L}(\mathcal{A}_\varphi)$), w is a witness that $M_1 \parallel M_2 \not\models \varphi$. In all other cases, however, it is not clear whether w should be classified as + or –.

In [5], the following heuristic is proposed. Let $B := \overline{M_1} \cup \mathcal{A}_\varphi$, where \overline{A} denotes complementation. Thus, runs of B either satisfy φ or are not in the behavior of M_1 and thus not in the behavior of M_1 running in parallel with any module A or

M_2 . Now, if $M_1 \parallel M_2 \models \varphi$ then $\mathcal{L}(M_2) \subseteq \mathcal{L}(B)$ as M_2 may only consist of words either satisfying φ or ones that are removed when intersecting with M_1 . If, on the other hand, $M_1 \parallel M_2 \not\models \varphi$, then there is a $w \in \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$ and $w \notin \mathcal{L}(A_\varphi)$, meaning that $w \in \mathcal{L}(M_1) \cap \mathcal{L}(M_2) \cap \overline{\mathcal{L}(A_\varphi)}$. Hence, $w \in \overline{\mathcal{L}(M_1)} \cup \mathcal{L}(A_\varphi) \cap \mathcal{L}(M_2)$. Thus, there is a w that is not in $\mathcal{L}(B)$ but in $\mathcal{L}(M_2)$.

In other words, B is a module that allows to either show or disprove that $M_1 \parallel M_2 \models \varphi$. Thus, answering membership queries according to B will eventually either show or disprove $M_1 \parallel M_2 \models \varphi$. While incrementally learning B , it is, however, expected, that one gets a hypothesis A smaller than B that satisfies (AG1) and (AG2) and thus shows that $M_1 \parallel M_2 \models \varphi$, or, that we get a word w witnessing that $M_1 \parallel M_2 \models \varphi$ does not hold.

The approach is sketched in Figure 4, where *ce x* is a shorthand for *counter example*.

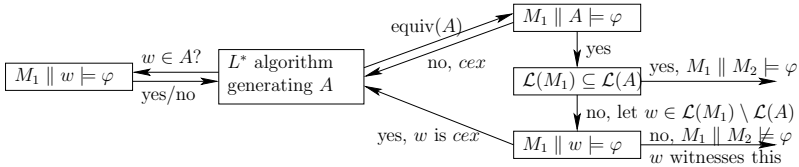


Fig. 4. Overview of compositional verification by learning assumptions

In [5], the approach has been worked out in the context of the verification tool NuSMV [20]. For this, a *symbolic* version of L^* has been developed that is based on BDDs [17] and carries out membership queries for sets of words rather than individual ones. It has been shown that the approach is beneficial for many examples. The success of the method, however, depends heavily on finding a suitable system A while learning B .

Instead of using Angluin’s L^* algorithm, one could think of using an inexperienced teacher answering membership queries by ? whenever a choice is possible. While the learning algorithm is computationally more expensive, smaller invariants A can be expected. It would be interesting to compare both approaches on real world examples.

In [18], the assume-guarantee reasoning for simulation conformance between finite state systems and specifications is considered. A non-circular assume-guarantee proof rule is considered, for which a weakest assumption can be represented canonically by a deterministic tree automaton (DTA). Then, learning techniques for DTA are developed and examined by verifying non-trivial benchmarks.

In [25], game semantics, counterexample-guided abstraction refinement, assume-guarantee reasoning and Angluin’s L^* algorithm are combined to yield a procedure for compositional verification of safety properties for fragments of sequential programs. For this, L^* is adapted to learn (regular) game strategies.

3.4 Learning Fixpoints, Regular Model Checking, and Learning Network Invariants

Assume that we are given a regular set of initial states $Init$ of a system to verify and a function Φ that computes for a given set W the set $\Phi(W)$ comprising of W together with successor states of W . Then, the set of reachable states is the least fixpoint $\lim_{n \rightarrow \infty} \Phi^n(W)$, where $\Phi^0(W) := Init$ and $\Phi^{n+1}(W) := \Phi(\Phi^n(W))$, for $n \geq 0$.

When proving properties for the set of reachable states, the typical approach would be to compute the (exact) set of reachable states by computing the minimal fixpoint using Φ , before starting to verify their properties. However, this approach may be problematic for two reasons:

- computing the fixpoint might be expensive, or
- the computation might not even terminate.

In regular model checking [2], for example, the set of initial states is regular (and represented by a finite automaton) and the set of successor states is computed by means of a transducer and thus is a regular set as well. There are, however, examples for which the set of reachable states is no longer regular. Thus, a straightforward fixpoint computation will not terminate.

A way out would be to consider a regular set of states over-approximating the set of reachable states.

Here, the idea of using learning techniques comes into play: Instead of computing iteratively the fixpoint starting from the initial states, one iteratively *learns* a fixpoint W . Clearly, one can stop whenever

- W is a fixpoint,
- W is a superset of $Init$, and
- W satisfies the property to verify.

For example, if one intends to verify a safety property, i.e., none of the reachable states is contained in the given bad states Bad , it suffices to find *any* fixpoint W that subsumes $Init$ and does not intersect with Bad .

This general idea has been worked out in different flavors for verifying properties of infinite-state systems: Verifying safety-properties of parameterized systems by means of *network invariants* has been studied in [35]. Applications of learning in *regular model checking* [2] for verifying safety properties [37,56] and liveness properties [57] can also be understood as a way to find suitable fixpoints. A further application for infinite state systems is that of verifying *FIFO automata* [55]. Let us understand the gist of these approaches.

Learning network invariants. One of the most challenging problems in verification is the *uniform verification of parameterized systems*. Given a parameterized system $S(n) = P[1] \parallel \dots \parallel P[n]$ and a property φ , uniform verification attempts to verify that $S(n)$ satisfies φ for every $n > 1$. The problem is in general undecidable [7]. One possible approach is to look for restricted families of systems for which the problem is decidable (cf. [28,22]). Another approach is

to look for sound but incomplete methods (e.g., explicit induction [29], regular model checking [39,51], or environment abstraction [23]).

Here, we consider uniform verification of parameterized systems using the heuristic of network invariants [60,43]. In simple words, a *network invariant* for a given finite system P is a finite system I that abstracts the composition of every number of copies of P running in parallel. Thus, the network invariant contains all possible computations of every number of copies of P . If we find such a network invariant I , we can solve uniform verification with respect to the family $S(n) = P[1] \parallel \dots \parallel P[n]$ by reasoning about I .

The general idea proposed in [60] and turned into a working method in [43], is to show by induction that I is a network invariant for P . The induction base is to prove that **(I1)** $P \sqsubseteq I$, for a suitable abstraction relation \sqsubseteq . The induction step is to show that **(I2)** $P \parallel I \sqsubseteq I$. After establishing that I is a network invariant we can prove **(P)** $I \models \varphi$, turning I into a *proper* network invariant with respect to φ . Then we conclude that $S(n) \models \varphi$ for every value of n .

Coming up with a proper network invariant is usually an iterative process. We start with divining a candidate for a network invariant. Then, we try to prove by induction that it is a network invariant. When the candidate system is non-deterministic this usually involves deductive proofs [42]². During this stage we usually need to refine the candidate until getting a network invariant. The final step is checking that this invariant is proper (by automatically model checking the system versus φ). If it is not, we have to continue refining our candidate until a proper network invariant is found. Coming up with the candidate network invariant requires great knowledge of the parameterized system in question and proving abstraction using deductive methods requires great expertise in deductive proofs and tools. Whether a network invariant exists is undecidable [60], hence all this effort can be done in vain.

In [35], a procedure searching systematically for a network invariant satisfying a given safety property is proposed. If one exists, the procedure finds a proper invariant with a minimal number of states. If no proper invariant exists, the procedure in general diverges (though in some cases it may terminate and report that no proper invariant exists). In the light of the undecidability result for the problem, this seems reasonable.

Network invariants are usually explained in the setting of *transition structures* [41]. However, the learning algorithms have been given in terms of DFAs (see Section 2). Thus, we explain the approach in the setting of checking safety properties of networks that are described in terms of (the parallel product of) DFAs: We assume that P is given as a DFA and abstraction is just language inclusion: $\mathcal{A} \sqsubseteq \mathcal{B}$ iff $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. A *safety property* is a DFA φ that accepts a *prefix-closed* language. Thus, a system \mathcal{A} satisfies φ , denoted by $\mathcal{A} \models \varphi$, iff $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\varphi)$. The *parallel operator* combines two given DFAs into a new one. Finally, a *projection operator* for $\mathcal{A} \parallel \mathcal{B}$ onto \mathcal{B} is a mapping $pr_{\mathcal{A} \parallel \mathcal{B}}^{\mathcal{B}} : \mathcal{L}(\mathcal{A} \parallel \mathcal{B}) \rightarrow \mathcal{L}(\mathcal{B})$ such that whenever $w \in \mathcal{L}(\mathcal{A} \parallel \mathcal{B})$ then for all \mathcal{B}' with $pr_{\mathcal{A} \parallel \mathcal{B}'}^{\mathcal{B}}(w) \in \mathcal{L}(\mathcal{B}')$ also

² For a recent attempt at mechanizing this step see [40].

$w \in \mathcal{L}(\mathcal{A} \parallel \mathcal{B}')$. In other words, (at least) the projection of w has to be removed from \mathcal{B} to (eventually) remove w from the parallel product.

The careful reader observes that the proper invariant I we are looking for is indeed a fixpoint of the operator $(P \parallel \cdot)$, which subsumes the words given by P and has empty intersection with the bad words given by $\mathcal{L}(\varphi)$. Thus, the following explanation can be understood as one way of learning fixpoints.

We now describe how to compute a proper network invariant in the case that one exists. For the rest of this section, we fix system P and a property automaton φ .

We only give an informal explanation, details can be found in [35]. We are using an unbounded number of *students* whose job it is to suggest possible invariants, one *teaching assistant* (TA) whose job is to answer queries by the students, and one *supervisor* whose job is to control the search process for a proper invariant. The search starts by the supervisor instructing one student to look for a proper invariant.

Like in Angluin's algorithm, every active student maintains a table (using $+$, $-$, and $?$) and makes it weakly closed and weakly consistent by asking the TA membership queries. The TA answers with either $+$, $-$, or $?$, as described below. When the table is weakly closed and consistent, the student translates the table to a sample O and this to a CSP problem. He solves the CSP problem using the SAT encoding (see Section 2.2). The solution with minimum range is used to form an automaton I that is proposed to the supervisor. The supervisor now checks whether I is indeed a proper invariant by checking (P), (I1), and (I2). If yes, the supervisor has found a proper invariant and the algorithm terminates with *proper invariant found*. If not, one of the following holds.

1. There is a string w such that $w \in \mathcal{L}(I)$ but $w \notin \mathcal{L}(\varphi)$,
2. There is a string w such that $w \in \mathcal{L}(P)$ but $w \notin \mathcal{L}(I)$,
3. There a string w such that $w \in \mathcal{L}(P \parallel I)$ but $w \notin \mathcal{L}(I)$.

In the first case, w should be removed from I . In the second case, the string w should be added to I . In these cases, the supervisor returns the appropriate string with the appropriate acceptance information to the student, who continues in the same manner as before.

In the last case, it is not clear, whether w should be added to I or removed from $P \parallel I$. For the latter, we have to remove the projection $pr_{P \parallel I}^I(w)$ from I . Unless w is listed negatively or $pr_{P \parallel I}^I(w)$ is listed positively in the table, both possibilities are meaningful. Therefore, the supervisor has to follow both tracks. She copies the table of the current student, acquires another student, and asks the current student to continue with w in I and the new student to continue with $pr_{P \parallel I}^I(w)$ not in I .

In order to give answers, the teaching assistant uses the same methods as the supervisor, however, whenever a choice is possible she just says $?$.

Choices can sometimes yield conflicts that are observed later in the procedure. Such a case reveals a conflicting assumption and requires the student to retire. If no working student is left, no proper invariant exists.

Clearly, the procedure sketched above finds a proper invariant if one exists. However, it consumes a lot of resources and may yield a proper invariant that is not minimal. It can, however, easily be adapted towards using only one student at a given time and stopping with a minimal proper invariant. Intuitively, the supervisor keeps track of the active students as well as the sizes of recently suggested automata. Whenever a student proposes a new automaton of size N , the supervisor computes the appropriate answer, which is either a change of the student's table or the answer *proper invariant found*. However, she postpones answering the student (or stopping the algorithm), gives the student priority N , and puts the student on hold. Then the supervisor takes a student that is on hold with minimal priority and sends the pre-computed instrumentation to the corresponding student. In case the student's instrumentation was tagged *proper invariant found* the procedure stops by printing the final proper invariant. Note that students always propose automata of at least the same size as before since the learning algorithm returns a *minimal* automaton conforming to the sample. Thus, whenever a proper invariant is found, it is guaranteed that the proper invariant is eventually reported by the algorithm, unless a smaller proper invariant is found before.

The formal details are given in [35].

Learning in regular model checking. In *regular model checking* [2] we are typically faced with a finite automaton *Init* encoding the initial states of an infinite-state system and a transducer τ , which yields for an automaton \mathcal{A} , an automaton $\tau(\mathcal{A})$ encoding the current and successor states of states given by \mathcal{A} . The set of reachable states is then given by the least fixpoint of *Init* under τ . However, the set of reachable states might not be regular, implying that a simple fixpoint computation does not terminate. Thus, for example, by so-called *acceleration* techniques, supersets of fixpoints are computed [1].

The same holds for the learning approaches in [37,56,57], in which algorithms employing an (experienced) teacher are used, in contrast to the approach of learning network invariants.

Clearly, when learning a fixpoint \mathcal{A} , equivalence queries are not difficult to answer: $\text{Init} \subseteq \mathcal{L}(\mathcal{A})$ and $\mathcal{L}(\tau(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{A})$ can easily be answered and, if applicable, a counterexample can be computed. For membership queries, the situation is more involved. In [37,56], the transducer is assumed to be *length preserving*, meaning that τ applied to some word w yields words w' of the same length. Thus, the fixpoint of w under τ can be computed in finitely many steps. Then, the following heuristic is used: Given a word w' of length n , check whether there is some w of length n such that w' is in the fixpoint of w . It has been reported, that this heuristic works well in practice [37,56]. However, it is not clear whether a regular fixpoint is found in this manner, if one exists.

In [57], verifying also *liveness* properties in the setting of regular model checking has been considered. For this, the set of reachable states *together* with information i on how many final states of a system encounter on a path of some

length j leaving s is learned. It has been shown that for the function computing successor states plus this additional information, a *unique* fixpoint exists. Clearly, liveness properties can be answered when the fixpoint is given. Furthermore, for a given (s, i, j) , it is easy to answer a membership query. Thus, the method works, provided a regular description for such a fixpoint exists.

Learning for verifying branching-time properties in the context of regular model checking asks for learning nested fixpoints and has been studied in [58].

3.5 Further Applications

Synthesizing interface specifications. Learning interface specifications for Java classes has been based on Angluin’s algorithm in [3]. The problem studied is to derive (a description) of the most general way to call the methods in a Java class while maintaining some safety property. In [3], the problem is tackled by abstracting the class, giving rise to a partial-information two-player game. As analyzing such games is computationally expensive, approximative solutions based on learning are considered.

Learning versus testing. In model-based testing [16], test suites are generated based on a model of the system under test (SUT). When no model is available, approximations of the SUT can be learned, the model can be analyzed, and used for test case generation. This approach, which is conceptually similar to black box checking, has been turned into a working method in [38].

In [10], the close relationship of learning techniques and test suites has been elaborated. In simple words, the following insight has been formalized and proved: if a conformance test suite is good enough to make sure that the SUT conforms to a given model, it should have enough information to identify the model. Likewise, if the observations of a system identify a single model, the observations should form a conformance test suite.

References

1. Abdulla, P.A., Jonsson, B., Nilsson, M., d’Orso, J.: Algorithmic improvements in regular model checking. In: Hunt, Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 236–248. Springer, Heidelberg (2003)
2. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004)
3. Alur, R., Cerný, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: Palsberg, J., Abadi, M. (eds.) POPL, pp. 98–109. ACM Press, New York (2005)
4. Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* 126(2), 183–235 (1994)

5. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 548–562. Springer, Heidelberg (2005)
6. Angluin, D.: Learning regular sets from queries and counterexamples. *Information and Computation* 75(2), 87–106 (1987)
7. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.* 22(6), 307–309 (1986)
8. Balcázar, J.L., Díaz, J., Gavaldá, R.: Algorithms for learning finite automata from queries: A unified view. In: *Advances in Algorithms, Languages, and Complexity*, pp. 53–72. Kluwer, Dordrecht (1997)
9. Baresi, L., Heckel, R. (eds.): FASE 2006 and ETAPS 2006. LNCS, vol. 3922, pp. 27–28. Springer, Heidelberg (2006)
10. Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B.: On the correspondence between conformance testing and regular inference. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 175–189. Springer, Heidelberg (2005)
11. Berg, T., Jonsson, B., Leucker, M., Saksena, M.: Insights to Angluin’s learning. In: *Proceedings of the International Workshop on Software Verification and Validation (SVV 2003)*. *Electronic Notes in Theoretical Computer Science*, vol. 118, pp. 3–18 (December 2003)
12. Berg, T., Jonsson, B., Raffelt, H.: Regular inference for state machines with parameters. In: Baresi and Heckel [9], pp. 107–121
13. Berg, T., Raffelt, H.: Model checking. In: Broy et al. [16]
14. Biermann, A.W., Feldman, J.A.: On the synthesis of finite-state machines from samples of their behaviour. *IEEE Transactions on Computers* 21, 592–597 (1972)
15. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M.: Replaying play in and play out: Synthesis of design models from scenarios by learning. In: Grumberg, O., Huth, M. (eds.) ETAPS 2007 and TACAS 2007. LNCS, vol. 4424, Springer, Heidelberg (2007)
16. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): *Model-Based Testing of Reactive Systems*. LNCS, vol. 3472. Springer, Heidelberg (2005)
17. Bryant, R.E.: Symbolic manipulation of boolean functions using a graphical representation. In: *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, pp. 688–694. IEEE Computer Society Press, Los Alamitos, CA (1985)
18. Chaki, S., Clarke, E.M., Sinha, N., Thati, P.: Automated assume-guarantee reasoning for simulation conformance. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 534–547. Springer, Heidelberg (2005)
19. Chow, T.S.: Testing software design modeled by finite-state machines. *IEEE Trans. on Software Engineering* 4(3), 178–187 (1978) (Special collection based on COMP-SAC)
20. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, Springer, Heidelberg (2002)
21. Clarke, E.M., Grumberg, O., Peled, D.A.: *Model Checking*. The MIT Press, Cambridge, Massachusetts (1999)

22. Clarke, E.M., Talupur, M., Touili, T., Veith, H.: Verification by network decomposition. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 276–291. Springer, Heidelberg (2004)
23. Clarke, E.M., Talupur, M., Veith, H.: Environment abstraction for parameterized verification. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 126–141. Springer, Heidelberg (2005)
24. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) ETAPS 2003 and TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
25. Dimovski, A., Lazic, R.: Assume-guarantee software verification based on game semantics. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 529–548. Springer, Heidelberg (2006)
26. Drewes, F., Högberg, J.: Learning a regular tree language from a teacher. In: Ésik, Z., Fülöp, Z. (eds.) DLT 2003. LNCS, vol. 2710, pp. 279–291. Springer, Heidelberg (2003)
27. D’Souza, D.: A logical characterisation of event clock automata. *International Journal of Foundations of Computer Science (IJFCS)* 14(4), 625–639 (2003)
28. Emerson, E., Kahlon, V.: Reducing model checking of the many to the few. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 236–254. Springer, Heidelberg (2000)
29. Emerson, E., Namjoshi, K.: Reasoning about rings. In: Proc. 22th ACM Symp. on Principles of Programming Languages (1995)
30. Gold, E.M.: Language identification in the limit. *Information and Control* 10, 447–474 (1967)
31. Gold, E.M.: Complexity of automaton identification from given data. *Information and Control* 37(3), 302–320 (1978)
32. Grinchtein, O., Jonsson, B., Leucker, M.: Inference of timed transition systems. In: 6th International Workshop on Verification of Infinite-State Systems. *Electronic Notes in Theoretical Computer Science*, vol. 138/4, Elsevier Science Publishers, Amsterdam (2004)
33. Grinchtein, O., Jonsson, B., Leucker, M.: Learning of event-recording automata. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, Springer, Heidelberg (2004)
34. Grinchtein, O., Jonsson, B., Pettersson, P.: Inference of event-recording automata using timed decision trees. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 435–449. Springer, Heidelberg (2006)
35. Grinchtein, O., Leucker, M., Piterman, N.: Inferring network invariants automatically. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, Springer, Heidelberg (2006)
36. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. In: Katoen, J.-P., Stevens, P. (eds.) ETAPS 2002 and TACAS 2002. LNCS, vol. 2280, p. 357. Springer, Heidelberg (2002)
37. Habermehl, P., Vojnar, T.: Regular model checking using inference of regular languages. *Electr. Notes Theor. Comput. Sci.* 138(3), 21–36 (2005)
38. Hungar, H., Niese, O., Steffen, B.: Domain-specific optimization in automata learning. In: Proc. 15th Int. Conf. on Computer Aided Verification (2003)
39. Jonsson, B., Nilsson, M.: Transitive closures of regular relations for verifying infinite-state systems. In: Schwartzbach, M.I., Graf, S. (eds.) ETAPS 2000 and TACAS 2000. LNCS, vol. 1785, Springer, Heidelberg (2000)

40. Kesten, Y., Piterman, N., Pnueli, A.: Bridging the gap between fair simulation and trace inclusion. *Information and Computation* 200(1), 35–61 (2005)
41. Kesten, Y., Pnueli, A.: Control and data abstraction: The cornerstones of practical formal verification. *Software Tools for Technology Transfer* 2(4), 328–342 (2000)
42. Kesten, Y., Pnueli, A., Shahar, E., Zuck, L.: Network invariants in action. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) *CONCUR 2002*. LNCS, vol. 2421, Springer, Heidelberg (2002)
43. Kurshan, R.P., McMillan, K.L.: A structural induction theorem for processes. *Information and Computation* 117(1), 1–11 (1995)
44. Lang, K.J.: Random dfa's can be approximately learned from sparse uniform examples. In: *COLT*, pp. 45–52 (1992)
45. Maler, O., Pnueli, A.: On the learnability of infinitary regular sets, *Esprit Basic Research Action No 3096* (1991)
46. Namjoshi, K.S., Treffer, R.J.: On the competeness of compositional reasoning. In: Emerson, E.A., Sistla, A.P. (eds.) *CAV 2000*. LNCS, vol. 1855, pp. 139–153. Springer, Heidelberg (2000)
47. Oliveira, A.L., Silva, J.P.M.: Efficient algorithms for the inference of minimum size dfas. *Machine Learning* 44(1/2), 93–119 (2001)
48. Oncina, J., Garcia, P.: Inferring regular languages in polynomial update time. In: de la Blanca, N.P., Sanfeliu, A., Vidal, E. (eds.) *Pattern Recognition and Image Analysis. Series in Machine Perception and Artificial Intelligence*, vol. 1, pp. 49–61. World Scientific, Singapore (1992)
49. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. In: Wu, J., Chanson, S.T., Gao, Q. (eds.) *Formal Methods for Protocol Engineering and Distributed Systems, FORTE/PSTV*, pp. 225–240. Kluwer, Beijing, China (1999)
50. Pena, J.M., Oliveira, A.L.: A new algorithm for the reduction of incompletely specified finite state machines. In: *ICCAD*, pp. 482–489 (1998)
51. Pnueli, A., Shahar, E.: Liveness and acceleration in parameterized verification. In: Emerson, E.A., Sistla, A.P. (eds.) *CAV 2000*. LNCS, vol. 1855, pp. 328–343. Springer, Heidelberg (2000)
52. Raffelt, H., Steffen, B.: Learnlib: A library for automata learning and experimentation. In: Baresi and Heckel [9], pp. 377–380
53. Thomas, W.: Automata on infinite objects. In: van Leeuwen, J. (ed.) *Handbook of Theoretical Computer Science*, ch. 4, vol. B, pp. 133–191. Elsevier Science Publishers B. V, Amsterdam (1990)
54. Trakhtenbrot, B., Barzdin, J.: *Finite automata: behaviour and synthesis*. North-Holland, Amsterdam (1973)
55. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Actively learning to verify safety for fifo automata. In: Lodaya, K., Mahajan, M. (eds.) *FSTTCS 2004*. LNCS, vol. 3328, pp. 494–505. Springer, Heidelberg (2004)
56. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Learning to verify safety properties. In: Davies, J., Schulte, W., Barnett, M. (eds.) *ICFEM 2004*. LNCS, vol. 3308, pp. 274–289. Springer, Heidelberg (2004)
57. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Using language inference to verify omega-regular properties. In: Halbwachs, N., Zuck, L.D. (eds.) *TACAS 2005*. LNCS, vol. 3440, pp. 45–60. Springer, Heidelberg (2005)

58. Vardhan, A., Viswanathan, M.: Learning to verify branching time properties. In: Redmiles, D.F., Ellman, T., Zisman, A. (eds.) ASE, pp. 325–328. ACM, New York (2005)
59. Vasilevski, M.P.: Failure diagnosis of automata. *Cybernetic* 9(4), 653–665 (1973)
60. Wolper, P., Lovinfosse, V.: Verifying properties of large sets of processes with network invariants. In: Sifakis, J. (ed.) *Automatic Verification Methods for Finite State Systems*. LNCS, vol. 407, pp. 68–80. Springer, Heidelberg (1990)

JACK — A Tool for Validation of Security and Behaviour of Java Applications*

Gilles Barthe¹, Lilian Burdy, Julien Charles¹, Benjamin Grégoire¹,
Marieke Huisman¹, Jean-Louis Lanet², Mariela Pavlova^{3,**},
and Antoine Requet²

¹ INRIA Sophia Antipolis, France

² Gemalto, France

³ Ludwig-Maximilians-Universität München, Germany

Abstract. We describe the main features of JACK (Java Applet Correctness Kit), a tool for the validation of Java applications, annotated with JML specifications. JACK has been especially designed to improve the quality of trusted personal device applications. JACK is fully integrated with the IDE Eclipse, and provides an easily accessible user interface. In particular, it allows to inspect the generated proof obligations in a Java syntax, and to trace them back to the source code that gave rise to them. Further, JACK provides support for annotation generation, and for interactive verification. The whole platform works both for source code and for bytecode, which makes it particularly suitable for a proof carrying code scenario.

1 Introduction

Motivation Over the last years, the use of trusted personal devices (TPD), such as mobile phones, PDAs and smart cards, has become more and more widespread. As these devices are often used with security-sensitive applications, they are an ideal target for attacks. Traditionally, research has focused on avoiding hardware attacks, where the attacker has physical access to the device to observe or tamper with it. However, TPD are more and more connected to networks and moreover, provide support to execute complex programs. This has increased the risk of logical attacks, which are potentially easier to launch than physical attacks (they do not require physical access, and are easier to replicate from one device to the other), and may have a huge impact. In particular, a malicious attacker spreading over the network and massively disconnecting or disrupting devices could have significant consequences.

An effective means to avoid such attacks is to improve the quality of the software deployed on the device. This paper describes JACK[†] (the Java Applet

* This work is partially funded by the IST programme of the European Commission, under the IST-2003-507894 *Inspired* and IST-2005-015905 *Mobius* projects.

** Research done while at INRIA Sophia Antipolis.

[†] See <http://www-sop.inria.fr/everest/soft/Jack/jack.html> for download instructions.

Correctness Kit), a tool that can be used to improve the quality of applications for TPD. Such devices typically implement the Java Virtual Machine (or one of its variations)². Therefore JACK is tailored to applications written in Java (bytecode). However, the described techniques are also relevant to other execution platforms for TPD.

Characteristics of JACK. JACK allows to verify Java applications that are annotated with the Java Modeling Language (JML)³. An advantage of using JML is that there is wide range of tools and techniques available that use JML as specification language, *i.e.*, for testing, simulation and verification (see [11] for an overview). We distinguish two kinds of verification: at runtime, using `jmlc`, or statically. Several tools provide static verification of JML-annotated programs, adopting different compromises between soundness, completeness and automation (Section 8 provides an overview of related work). JACK implements a weakest precondition calculus, that automatically generates proof obligations that can be discharged both by automatic and interactive theorem provers. The automatic prover that is used is Simplify [22], the interactive theorem prover that is used is Coq [39].

The development of the JACK tool started in 2002 at the formal methods research laboratory of the French smart card producer Gemplus (now part of gemalto). Successful case studies with ESC/Java [17] and the LOOP tool [8] on an electronic purse smart card application [10] had sufficiently demonstrated that verification of JML annotations could help to increase the quality of smart card applications. However, the existing tools were either not precise enough, or too cumbersome to use to expose application developers to them. The JACK tool was designed to overcome these problems, in particular via the integration of JACK within the IDE Eclipse⁴, and the development of a special JACK perspective.

In 2003, the tool has been transferred to the Everest project at INRIA Sophia Antipolis, and been further developed within this team since then. The other features of JACK described in this paper have been developed after this transfer.

The main characteristics of JACK that distinguish it from other static verification tools are the following:

- full integration within Eclipse IDE, including the development of a special JACK perspective that allows to inspect the different proof obligations, and from where in the code they originate;
- implementation of annotation generation algorithms: to generate “obvious” annotations, and to encode high-level security properties;
- support for verification of bytecode programs; and

² The standard Java set-up for TPD is the Connected Limited Device Configuration, see <http://java.sun.com/products/cldc/>, together with the MIDP profile, see <http://java.sun.com/products/midp/>

³ See <http://www.jmlspecs.org>

⁴ See <http://www.eclipse.org>

- support for interactive verification, by the development of an interface and tactics for Coq and by use of the `native` construct, that allows to link JML specifications with the logic of the underlying theorem prover.

This paper illustrates how these characteristics make JACK particularly suited for the development of secure applications for TPD.

Application Scenarios. JACK provides different kinds of support for the application developer, ranging from the automatic verification of common security properties to the interactive verification of complex functional specifications.

To support the automatic verification of high-level security properties, JACK provides an algorithm to automatically generate annotations encoding such properties, and to weave and propagate these in the application. These annotations give rise to proof obligations, whose discharge (typically automatic) guarantees adherence to the security policy. Since JACK also provides support for the verification of bytecode, and allows to compile source code level JML annotations into bytecode level specifications (written in the Bytecode Modeling Language (BML) [12]), this enables a proof carrying code scenario [35]. In such a scenario, the applications come equipped with a specification and a proof that allow the client to establish trust in the application. Since the applications usually are shipped in bytecode format, also the specification and the verification process need to be defined at this level. This scenario is even further facilitated by the fact that the compiler from JML to BML provided by JACK basically preserves the generated proof obligations (see also [6]). Thus, a software developer can verify its applications at source code level, and ship them with compiled bytecode level specifications and proofs. Notice that, provided the proof obligations can be discharged automatically, this whole process is automatic.

However, as JACK is a general-purpose tool, it can be also be used to verify complex functional-behaviour specifications. For this, it provides advanced support for specification development and interactive verification. Because of the tight integration with Eclipse, the developer does not have to change tools to validate the application. A special JACK view is provided, that allows to inspect the generated proof obligations in different views (in a Java-like syntax, or in the language of the prover). Moreover, syntax colouring of the original source code allows to see to which parts of the application and specification the proof obligation relates. Further, JACK can generate “obvious” annotations that are easy to forget, in particular preconditions that are sufficient to avoid runtime exceptions. This helps to overcome one of the major drawbacks of using JML-like annotations for specifications, namely that writing annotations is labour-intensive and error-prone. Finally, to support interactive verification, several advanced Coq tactics have been developed, and a Coq editor has been integrated into Eclipse. In addition, to be able to write expressive specifications, a `native` construct has been proposed for JML, that allows to link JML constructs directly with the logic of the underlying prover. This allows to develop the theory about these constructs directly in the logic of the theorem prover, which makes specification and verification simpler.

Overview of the Paper. The next section gives a quick overview of the relevant JML features. Section 3 briefly outlines the general architecture of JACK, while Section 4 focuses on its user interface. Section 5 describes the different annotation generation algorithms that JACK implements. Section 6 presents the bytecode subcomponents of JACK, while Section 7 explains the features that JACK provides to support interactive verification. Finally, Section 8 concludes and discusses how this work will be continued.

Parts of the results described in this paper have been published elsewhere: [14] describes the general architecture of JACK, [37] the annotation generation algorithm for security policies, [13] the framework for the verification of bytecode, and [16] the native construct. However, this is the first time a complete overview of JACK and its main features are given in a single paper.

2 A Quick Overview of JML

This section gives a short overview of JML, by means of an example. Throughout the rest of this paper, we assume the reader is familiar with JML, its syntax and semantics. For a detailed overview of JML we refer to the reference manual [31]; a detailed overview of the tools that support JML can be found in [11]. Notice that JML is designed to be a general specification language that does not impose any particular design method or application domain [29].

To illustrate the different features of JML, Figure 1 shows a fragment of a specification of class `QuickSort`. It contains a public method `sort`, that sorts the array stored in the private field `tab`. Sorting is implemented via a method `swap`, swapping two elements in the array, and a private method `sort`, that actually implements the quicksort algorithm.

In order not to interfere with the Java compiler, JML specifications are written as special comments (tagged with `@`). Method specifications contain preconditions (keyword `requires`), postconditions (`ensures`) and frame conditions (`assignable`). The latter specifies which variables *may* be modified by a method. In a method body, one can annotate all statements with an `assert` predicate and loops also with invariants (`loop_invariant`), and variants (`decreases`). One can also specify class invariants, *i.e.*, properties that should hold in all visible states of the execution, and constraints, describing a relation that holds between any two pairs of consecutive visible states (where visible states are the states in which a method is called or returned from).

The predicates in the different conditions are side-effect free Java boolean expressions, extended with specification-specific keywords, such as `\result`, denoting the return value of a non-void method, `\old`, indicating that an expression should be evaluated in the pre-state of the method, and the logical quantifiers `\forall` and `\exists`. Re-using the Java syntax makes the JML specifications easily accessible to Java developers.

JML allows further to declare special specification-only variables: logical variables (with keyword `model`) and so-called `ghost` variables, that can be assigned to in special `set` annotations.

```

public class QuickSort {
    private int [] tab;

    public QuickSort(int[] tab) {this.tab = tab;}

    /*@ requires (tab != null) ;
       @ assignable tab[0 .. (tab.length - 1)];
       @ ensures (\forallall int i, j; 0 <= i && i <= (tab.length - 1) ==>
       @           0 <= j && j <= (tab.length - 1) ==>
       @           i < j ==> tab[i] <= tab[j]) &&
       @           (\forallall int i; 0 <= i && i <= (tab.length - 1) ==>
       @           (\exists int j; 0 <= j && j <= (tab.length - 1) &&
       @           \old(tab[j]) == tab[i])); @*/
    public void sort() {if(tab.length > 0) sort(0, tab.length - 1);}

    /*@ requires (tab != null) && (0 <= i) && (i < tab.length) &&
       @           (0 <= j) && (j < tab.length);
       @ assignable tab[i], tab[j];
       @ ensures tab[i] == \old(tab[j]) && (tab[j] == \old(tab[i])); @*/
    public void swap(int i, int j) { ... }

    private void sort(int lo, int hi) { ... }
}

```

Fig. 1. Fragment of class `QuickSort` with JML annotations

Figure 1 specifies that method `sort` sorts the array `tab` from low to high, and all elements that occurred in the array initially also occur in its afterwards, and that method `swap` swaps the contents of the array at positions `i` and `j`.

3 General Architecture of JACK

This section describes the general architecture of JACK, and how it aims at a high level of precision. The next section then discusses how JACK has been made accessible to application developers by integration within the IDE Eclipse, and the development of the special JACK perspective. For the development of the JACK architecture, the main design principles were the following:

- integration within a widely-used IDE, so that developers do not have to learn a new environment, and do not have to switch between tools;
- automatic generation of proof obligations by implementation of a weakest precondition (wp) calculus;
- proof obligations are first-order logic formulae; and
- prover independence, *i.e.*, proof obligations for a single application can be verified with different provers.

⁵ Note that this specification does not require that the final value of `tab` is a sorted permutation of its initial value. However, this could be expressed in JML as well.

The wp-calculus that is implemented is a so-called “direct” calculus, meaning that it works directly on an AST representation of the application, and it does not use a transformation into guarded commands, as is done by *e.g.*, ESC/Java. The wp-calculus is based on the classical wp-calculus developed by Dijkstra [23], but adapted to Java by extending it with side-effects, exceptions and other abrupt termination constructs (*cf. e.g.*, [28]). Method invocations are abstracted by their specifications, since we want verification to be modular. This direct wp-calculus has the advantage that it is easy to generate proof obligations for each path through a method, and then to connect the proof obligation with the path through the method that gave rise to this particular proof obligation (to achieve this, also some program flow information is associated to each proof obligation). This connection makes the understanding of the generated proof obligations easier. Another advantage of this approach is that the algorithms for annotation generation as described below in Section 5 could make direct use of the weakest precondition infrastructure. A drawback of this approach is that the size of the generated proof obligation may be exponential in the size of the code fragment being checked [27].

To avoid this blow up in the size of the proof obligation, and to ensure that proof obligations can be generated automatically, JACK uses several new specification constructs, introduced in [14]: loops can be annotated with frame conditions (`loop_modifies`) and exceptional postconditions (`loop_exsures`), and any code block can be specified with a block specification (similar to a method specification). The loop frame condition is used in the the wp-calculus to make a universal quantification over the loop invariant when generating the appropriate proof obligations. Block specifications and `loop_exsures` clauses improve readability and reduce the number of proof obligations, because they reduce the number of paths through a method that have to be considered.

JACK generates its proof obligations in an abstract formula language, representing first-order logic formulae. It is straightforward to translate the abstract formulae into a proof obligation for a particular prover. Adding a new prover as a plug-in to the tool is simple: one develops a background theory formalising Java’s type system and memory model, and one defines how the abstract formulae are translated into concrete proof obligations for this particular prover. Initially, JACK was designed to use the AtelierB prover [1], now Simplify and Coq are the best supported back-end provers for JACK.

4 JACK’s User Interface

One of the features that distinguish JACK from other program verification tools is the integration in the IDE Eclipse. This ensures a seamless integration of formal methods in the application development process: the application developer does not have to learn the peculiarities of a new tool, and does not have to switch tools to apply formal verification techniques.

The integration in Eclipse consists of two parts: an extension of the standard Java perspective with special JACK-related actions (checking a specification,

calling an automatic prover *etc.*), and a special JACK perspective to inspect the generated proof obligations.

4.1 Extension of the Java Perspective in Eclipse

The standard Java perspective of Eclipse is extended with several JACK-specific features. Menus are added to set the defaults for the different specification constructs. Further, there are buttons and menu-options to “compile” a JML specification, (*i.e.*, type check and generate proof obligations), call an automatic prover on all the generated proof obligations (either Simplify or a special Coq tactic), or change to the special JACK perspective.

Checking the JML specification is not done in a background mode, while editing the file (as is done for the type checking of Java); instead the user has to launch this action explicitly. At the time this interface was developed, adding such automatic checks required too many changes to the internals of Eclipse, which were not default available. However, in the mean time such a feature has been developed within the JMLeclipse project⁶. This project also provides syntax highlighting of JML specifications in Eclipse’s Java perspective. All this could be integrated with the JACK interface.

Finally, another important constraint is the interface’s responsiveness. An IDE is supposed to be used interactively, and the developer should never have to wait long for a result. Proof obligation generation is no problem for this, but calling an automatic prover on the generated proof obligations can take a significant amount of time. Therefore, the prover is called in a non-blocking way, launching a special window that allows to see the progress of the task.

4.2 A Proof Obligation Inspection Perspective

An important feature of JACK is that one can inspect the different generated proof obligations. Moreover, one does not have to understand the specific specification language of the prover that is being used; instead the proof obligations can be viewed in a Java/JML-like syntax (but of course, one can also choose to see the proof obligations as they are generated for a specific theorem prover).

Figure 2 shows the inspection of a proof obligation for the method `sort` in the `QuickSort` example of Figure 1. The left upper windows allows one to browse the proof obligations for the current class. Proven obligations are ticked, the others are marked with a cross. The right window shows the original source code, where the path through the code that corresponds to the current proof obligation is coloured, together with the relevant part of the method specification. Different colours are used to indicate different cases, *i.e.*, to distinguish normal from exceptional execution, and to mark that extra information, such as a method specification, or the result of a conditional expression, is available. For example, in Figure 2 one sees the specification of the private `sort` method in a pop-up box, used in the public `sort` method.

⁶ See <http://jmleclipse.projects.cis.ksu.edu/>

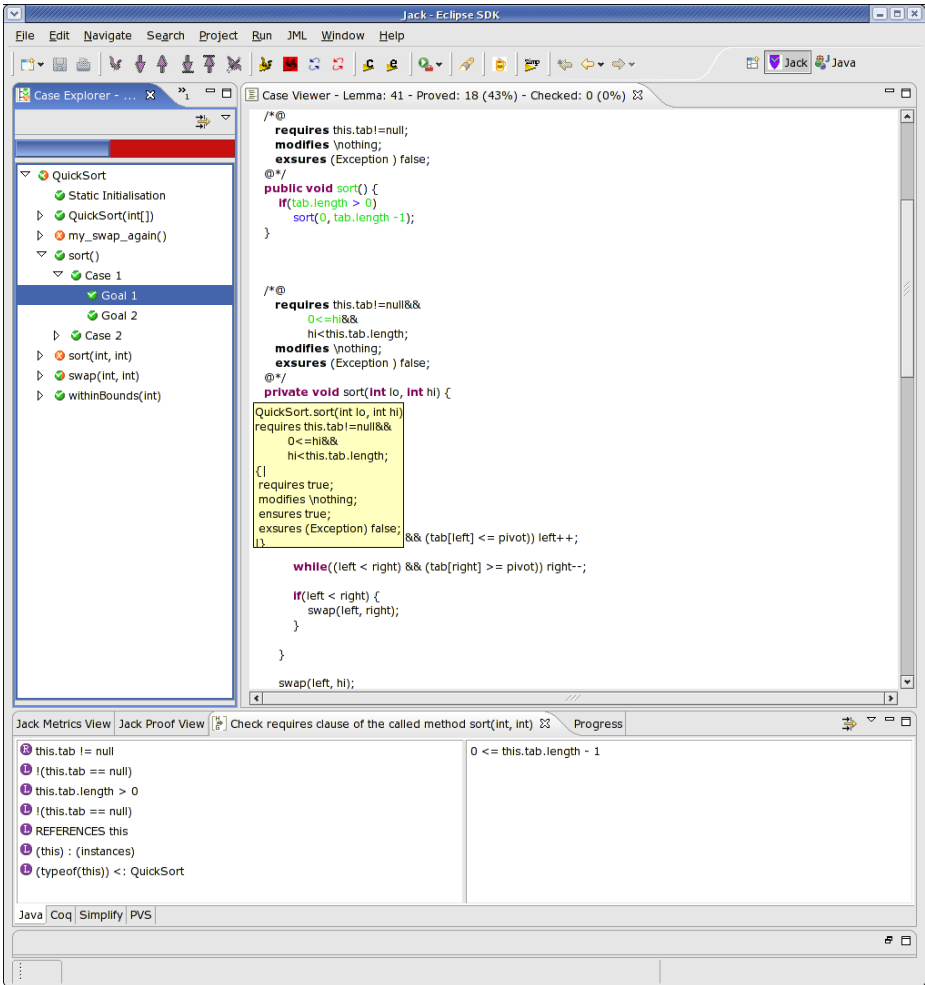


Fig. 2. JACK’s proof obligation inspection perspective

The bottom window shows the proof obligation: the left half contains the hypotheses, marked with letters indicating their origin, *e.g.*, a hypothesis marked R originates from the method’s requires clause, while a hypothesis marked L is derived from local declarations within the method. The right half of the window shows the actual goal that has to be proven. The window name highlights once again that this proof obligation originates from the postcondition. Finally, notice that the proof obligation is displayed in Java syntax, but buttons are available to change to Coq, Simplify or PVS syntax.

The user can use the proof obligation inspection view to inspect the different (unproven) proof obligations, and to launch different (interactive or specialised) provers to prove the remaining proof obligations.

```

/*@ requires this.tab!=null;
    signals (Exception) false;
  @*/
public void sort() {if(tab.length > 0) sort(0, tab.length -1);}

/*@ requires this.tab!=null && 0<=j && j<this.tab.length &&
    0<=i && i<this.tab.length;
    signals (Exception) false;
  @*/
public void swap(int i, int j) {int tmp; tmp = tab[i];
                                tab[i] = tab[j]; tab[j] = tmp;}

```

Fig. 3. Obvious annotations generated for a fragment of class `QuickSort`

5 Generating JML Annotations

While JML is easily accessible to Java developers, actually writing the specifications of an application is labour-intensive and error-prone, as it is easy to forget some annotations. There exist tools which assist in writing these annotations, *e.g.*, Daikon [25] and Houdini [26] use heuristic methods to produce annotations for simple safety and functional invariants. However, these tools cannot be guided by the user—they do not require any user input—and in particular cannot be used to generate annotations from realistic security policies.

Within JACK, we have implemented several algorithms to generate annotations. We can distinguish two goals for annotation generation. The first is to reduce the burden of annotation writing by generating as much “obvious” annotations as possible. Given an existing, unannotated, application, one first generates these obvious annotations automatically, before developing the more interesting parts of the specification. The second goal is to encode high-level properties by encoding these with simple JML annotations, that are inserted at all appropriate points in the application, so that they can be checked statically. JACK implements algorithms for both goals, as described in this section.

5.1 Generation of Preconditions

JACK implements an algorithm to generate “obvious” minimal preconditions to avoid null-pointer and array-out-of-bounds exceptions. This algorithm re-uses the implementation of the wp-calculus: it computes the weakest precondition for the specification `signals (NullPointerException) false;` (resp. `signals (ArrayIndexOutOfBoundsException) false;`) and inserts this as annotations in the code.

As an example, Figure 3 shows the annotations that are generated for some methods of the class `QuickSort` of Figure 1. It is important to realise that the specifications that are generated might not be very spectacular, but that they are generated *automatically*.

The annotation generation could be further improved by applying a simple analysis on the generated annotations. Often it is the case that the preconditions that are generated for the fields of the class are the same for (almost) all methods. In that case, this condition is likely to be a class invariant, and instead of generating a precondition for each method, it would be more appropriate to generate a single class invariant. For example, for the class `QuickSort`, this would produce an annotation `invariant this.tab!=null;`.

5.2 Encoding of Security Policies

Another difficulty when writing annotations is that a conceptually simple high-level property can give rise to many different annotations, scattered through the code, to encode this property. This is typically the case for many security policies. Current software practice for the development of applications for trusted personal devices is that security policies give rise to a set of *security rules* that should be obeyed by the implementation. Obedience to these rules is established by manual code inspection; however it is desirable to have tool support for this, because a typical security property may involve several methods from different classes. Many of the security rules can be formalised as simple automata, which are amenable to formal verification. Therefore, we propose a method that given a security rule, automatically annotates an application, in such a way that if the application respects the annotations then it also respects the security policy. Thus, it is not necessary for the user to understand the generated annotations, he just has to understand the security rules.

The generation of annotations proceeds in two phases: first we generate core-annotations that specify the behaviour of the methods directly involved, and next we propagate these annotations to all methods directly or indirectly invoking the methods that form the core of the security policy. The second phase is necessary because we are interested in static verification. The annotations that we generate all use only JML static ghost variables; therefore the properties are independent of the particular class instances available.

As a typical example of the kind of security rules our approach can handle, we consider the atomicity mechanism in Java Card (Java for smart cards) ([\[37\]](#) gives more examples of such security rules). A smart card does not include a power supply, thus a brutal retrieval from the terminal could interrupt a computation and bring the system in an incoherent state. To avoid this, the Java Card specification prescribes the use of a transaction mechanism to control synchronised updates of sensitive data. A statement block surrounded by the methods `beginTransaction()` and `commitTransaction()` can be considered atomic. If something happens while executing the transaction (or if `abortTransaction()` is executed), the card will roll back its internal state to the state before the transaction was begun. To ensure the proper functioning and prevent abuse of this mechanism, applications should respect for example the following security rules.

No nested transactions. Only one level of transactions is allowed.

No exception in transaction. All exceptions that may be thrown inside a transaction, should also be caught inside the transaction.

Bounded retries. No pin verification may happen within a transaction.

The second rule ensures that a transaction will always be closed; if the exception would not be caught, `commitTransaction` would not be executed. The last rule avoids the possibility to abort the transaction every time a wrong pin code has been entered. As this would roll back the internal state to the state before the transaction was started, this would also reset the retry counter, thus allowing an unbounded number of retries. Even though the specification of the Java Card API prescribes that the retry counter for pin verification cannot be rolled back, in general one has to check this kind of properties.

Such properties can be easily encoded with automata, describing in which states a certain method is allowed to be called. Based on this automata, we then generate core-annotations. For example, the atomicity properties above give rise to core-annotations for the methods related to the transaction mechanism declared in class `JCSYSTEM` of the Java Card API. A static ghost variable

```
/*@ static ghost int TRANS == 0; @*/
```

is declared, that is used to keep track of whether there is a transaction in progress. To specify the **No nested transactions** property, the core-annotations for method `beginTransaction` are the following.

```
/*@ requires TRANS == 0;
   @ assignable TRANS;
   @ ensures TRANS == 1; @*/
public static native void beginTransaction()
    throws TransactionException;
```

Similar annotations are generated for `commitTransaction` and `abortTransaction` ([\[37\]](#) also describes the generated core-annotations for the other properties). After propagation, these annotations are sufficient to check for the absence of nested transactions. To understand why propagation is necessary, suppose we are checking the **No nested transactions** property for an application, containing the following fragment (where `m` does not call any other methods, and does not contain any set-annotations).

```
void m() { ... // some internal computations
           JCSYSTEM.beginTransaction();
           ... // computations within transaction
           JCSYSTEM.commitTransaction(); }
```

When applying static verification on this code fragment, the core-annotations for `beginTransaction` will give rise to a proof obligation that the precondition of method `m` implies that there is no transaction in progress, *i.e.*, `TRANS == 0` (since `TRANS` is not modified by the code that precedes the call to `beginTransaction`).

The only way this proof obligation can be established is if the precondition of `beginTransaction` is propagated as a precondition for method `m`. In contrast, the precondition for `commitTransaction` (`TRANS == 1`) does not have to be propagated to the specification of `m`; instead it has to be established by the postcondition of `beginTransaction`, because the variable `TRANS` is modified by this method.

In a similar way, the postcondition for the method `commitTransaction` is propagated to the postcondition of method `m`. This information can then be used for the verification of yet another method, that contains a call to method `m`.

The propagation method not only propagates preconditions and normal and exceptional postconditions, it also propagates assignable clauses. We have shown that the algorithm that we use corresponds to an abstract version of the wp-calculus (where we only consider static variables). We have exploited this correspondence in the implementation, by re-using the wp-calculus infrastructure to implement the propagation algorithm. For a more formal treatment of the propagation algorithm, and the correspondence statement, we refer to [37].

To illustrate the effectiveness of our approach, we tested our method on several industrial smart card applications, including the so-called Demoney case study, developed as a research prototype by Trusted Logic⁷, and the PACAP case study [9], developed by Gemplus. Both examples have been explicitly developed as test cases for different formal techniques, illustrating the different issues involved when writing smart card applications. We used the core-annotations as presented above, and propagated these throughout the applications. For both applications we found that they contained no nested transactions, and that they did not contain attempts to verify pin codes within transactions. However, in the PACAP application we found transactions containing uncaught exceptions. All proof obligations generated *w.r.t.* these properties are trivial and can be discharged immediately. However, to emphasise the usefulness of having a tool for generating annotations: we encountered cases where a single transaction gave rise to twenty-three annotations in five different classes. When writing these annotations manually, it is all too easy to forget some.

6 Specification and Verification of Bytecode

JACK allows one to verify applications not only at source code level, but also at bytecode level. This is in particular important to support proof carrying code [35], where bytecode applications are shipped together with their specification and a correctness proof. However, the possibility to verify bytecode also has an interest on its own: sometimes security-critical applications are developed directly at bytecode level, in order not to rely on the correctness of the compiler. To be able to formally establish the correctness of such an application, one needs support to verify bytecode directly.

This section describes the different parts in the bytecode subcomponent of JACK. First, we present a specification language tailored to bytecode, and we

⁷ See <http://www.trusted-logic.com>

```

predicate ::= ...
unary-expr-not-plus-minus ::= ...
    | primary-expr [primary-suffix]...
primary-suffix ::= . ident | ( [expression-list] ) | [ expression ]
primary-expr ::= #natural           % reference in the constant pool
    | lv[natural]                   % local variable
    | bml-primary
    | constant | super | true | false | this | null | (expression) | jml-primary
bml-primary ::= cntr                % counter of the operand stack
    | st(additive-expr)             % stack expressions
    | length(expression)           % array length

```

Fig. 4. Fragment of grammar for BML predicates and specification expressions

specify how these specifications can be encoded in the class file format. Our specification language, called BML for Bytecode Modeling Language [12], is the bytecode cousin of JML. Second, we define and implement a compiler from JML to BML specifications. Such a compiler is in particular useful in a proof carrying code scenario, where the application developer can verify the application at (the more intuitive) source code level, and then compile both the application and the specification to bytecode level. Finally, we also define a verification condition generator for bytecode applications annotated with BML, implementing a wp-calculus for bytecode. This allows to generate the proof obligations for a bytecode application to satisfy its BML specification.

6.1 A Specification Language for Bytecode: BML

BML has basically the same syntax as JML with two exceptions:

1. specifications are not written directly in the program code, they are added as special attributes to the bytecode; and
2. the grammar for expressions only allows bytecode expressions.

Figure 4 displays the most interesting part of the grammar for BML predicates, defining the syntax for primary expressions and primary suffixes⁸. Primary expressions, followed by zero or more primary suffixes, are the most basic form of expressions, formed by identifiers, bracketed expressions *etc.*

Since only bytecode expressions can be used, all field names, class names *etc.*, are replaced by references to the constant pool (a number, preceded by the symbol #), while registers are used to refer to local variables and parameters. The grammar also contains several bytecode specific keywords, such as `cntr`, denoting the stack counter, `st(e)` where *e* is an arithmetic expression, denoting the *e*th element on the stack, and `length(a)`, denoting the length of array *a*. In addition, the specification-specific JML keywords are also available.

To show a typical BML specification, Figure 5 presents the BML version of the JML specification of method `swap` in Figure 1. Notice that the field `tab` has

⁸ See <http://www-sop.inria.fr/everest/BML> for the full grammar.

```

requires this.#14 != null && 0 <= lv[1] && lv[1] < length(this.#14) &&
    0 <= lv[2] && lv[2] < length(this.#14) && true
assignable this.#14.[lv[2]],this.#14[lv[1]]
ensures this.#14[lv[1]] == \old(this).\old(#14)[\old(lv[2])] &&
    this.#14[lv[2]] == \old(this).\old(#14)[\old(lv[1])]
0 aload_0
1 getfield #14
4 iload_1
5 iaload
6 istore_3
7 aload_0
8 getfield #14
11 iload_1
12 aload_0
13 getfield #14
16 iload_2
17 iaload
18 iastore
19 aload_0
20 getfield #14
23 iload_2
24 iload_3
25 iastore
26 return

```

Fig. 5. Bytecode + BML specification for method `swap` in class `QuickSort`

been assigned the number 14 in the constant pool, and that it is always explicitly qualified with `this` in the specification. In the bytecode the variable `this` is stored in `lv[0]` (thus it can be accessed by `aload_0`). The method's parameters `i` and `j` are denoted by the expressions `lv[1]` and `lv[2]`, respectively. Notice further that the BML specification directly corresponds to the original JML specification.

6.2 Encoding BML Specifications in the Class File Format

To store BML specifications together with the bytecode it specifies, we encode them in the class file format. The Java Virtual Machine Specification [32] prescribes the mandatory elements of the class file: the constant pool, the field information and the method information. User-specific information can be added to the class file as special user-specific attributes ([32, §4.7.1]). We store BML specifications in such user-specific attributes, in a compiler-independent format. The use of special attributes ensures that the presence of BML annotations does not have an impact on the application's performance, *i.e.*, it will not slow down loading or normal execution of the application.

```

Ghost_Field_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 fields_count;
  { u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
  } fields[fields_count]; }

```

Fig. 6. Format of attribute for ghost field declarations

For each class, we add the following information to the class file:

- a second constant pool which contains constant references for the BML specification expressions;
- an attribute with the ghost fields used in the specification;
- an attribute with the model fields used in the specification;
- an attribute with the class invariants (both static and object); and
- an attribute with the constraints (both static and object).

Apart from the second constant pool, all extra class attributes basically consist of the name of the attribute, the number of elements it contains, and a list with the actual elements. As an example, Figure 6 presents the format of the ghost field attribute. This should be understood as follows: the name of the attribute is given as an index into the (second) constant pool. This constant pool entry will be representing a string "Ghost_Field". Next we have the length of the attribute and the number of fields stored in the attribute. The **fields** table stores all ghost fields. For each field we store its access flag (*e.g.*, **public** or **private**), and its name and descriptor index, both referring to the constant pool. The first must be a string, representing the (unqualified) name of the variable, the latter is a field descriptor, containing *e.g.*, type information. The tags **u2** and **u4** specifies the size of the attribute, 2 and 4 bytes, respectively. The format of the other attributes is specified in a similar way (see [36] for more details).

6.3 Compiling JML Specifications into BML Specifications

We have implemented a compiler from JML specifications into BML specifications – stored in the class file. The JML specification is compiled separately from the Java source code. In fact, the compiler takes as input an annotated Java source file *and* the class file produced by a non-optimising compiler with the debug flag set.

From the debug information, we use in particular the **Line_Number_Table** and the **Local_Variable_Table** attributes. The **Line_Number_Table** links line numbers in the source code with the bytecode instructions, while the **Local_Variable_Table** describes the local variables that appear in a method.

JML specifications are compiled into BML specifications in several steps:

1. compilation of ghost and model field declarations;
2. linking and resolving of source data structures to bytecode structures;

3. locating instructions for annotation statements; this information is added as a special **index** entry in the attribute (a heuristic algorithm is used to find the entry point of a loop, for more details see [36]);
4. compilation of JML predicates, taking into account that not all source code level primitive types are directly supported at bytecode level; and
5. generation of user-specific class attributes.

6.4 Verification of Bytecode

To generate proof obligations, we have implemented a wp-calculus for bytecode in JACK. Just as the source code level wp-calculus, it works directly on the bytecode; the program is not transformed into a guarded command format. Again, this has the advantage that we can easily trace proof obligations back to the relevant bytecode and BML fragment.

The JACK implementation supports all Java bytecode sequential instructions, except for floating point arithmetic instructions and 64 bit data (**long** and **double**). Thus in particular, it handles exceptions, object creation, references and subroutines. The calculus is defined over the method’s control flow graph.

The verification condition generator proceeds as follows. For each method proof obligations are generated for each execution path by applying the weakest predicate transformer to every instruction where the method might end (*i.e.*, **return** or **athrow** instructions), and at each loop exit point. The wp-calculus then follows control flow backwards, until it reaches the entry point instruction.

The weakest precondition transformer takes three arguments: the instruction for which we calculate the precondition, the instruction’s normal postcondition ψ and the instruction’s exceptional postcondition ϕ^{Exc} . For the full wp-calculus for BML-annotated bytecode, and its soundness proof we refer to [36]. Here we show as an example the wp-rule for the instruction load_i .

$$\text{wp}(\text{load}_i, \psi, \psi^{Exc}) = \psi[\text{cntr} \leftarrow \text{cntr} + 1][\text{st}(\text{cntr} + 1) \leftarrow \text{lv}[i]]$$

Since the load_i instruction will always terminate normally, only the normal postcondition is involved, after updating it to reflect the changes that are made to the stack, *i.e.*, the value that was stored in the local variable register $\text{lv}[i]$ is now at the top of the stack (at position $\text{st}(\text{cntr} + 1)$), and the stack counter is increased.

Finally we would like to remark that there is a close correspondence between the proof obligations generated by JACK at source code level, and the proof obligations that are generated once the application and the specification are compiled at bytecode level (provided that the application is compiled with a non-optimising compiler): modulo names and the handling of shorts, bytes and boolean values, the proof obligations are equivalent. This means that proofs for proof obligations at source code level can be re-used for proof obligations at bytecode level (see also [6] for a compilation of source code level proofs to bytecode level proofs). This is in particular important for the proof carrying code

scenario [35], where the code producer develops a proof at source code level, and then ships bytecode level application and specification. Modulo the necessary re-namings, the proofs can be shipped directly, and the code client can verify these using a verification condition generator at bytecode level.

7 Support for Interactive Verification

When verifying complex functional behaviour specifications, automatic provers often fail to solve the proof obligations. In that case, the user can instead try to solve the proof obligation interactively (or, in case the proof obligations is unprovable, analyse it thoroughly to find the source of the error). JACK provides support for interactive verification using the Coq proof assistant [19].

This section first discusses the special features of JACK’s Coq plug-in to support interactive verification, and the special Coq editor integrated in Eclipse, then it presents JACK’s specific annotation keyword for interactive verification: the `native` keyword.

7.1 The Coq Plug-In

Proof readability and proof re-usability is crucial in interactive verification, in contrast to automatic verification where proof obligations are simply sent to the automatic prover and it is of no importance whether the proof obligation is human-readable or not. Therefore we developed a set of facilities for pretty printing proof obligations, to reuse the proofs – in particular to allow replaying the proofs when the specifications have changed – and for proof construction.

JACK uses short variable names in proof obligations as much as possible, but in case of ambiguity long variable names are used. Basically, JACK generates all variable names for all proof obligations of one file in one go. However, for interactive verification the variables only have to be considered within the scope of a single proof obligation, thus short variable names can be used more often. Therefore, the Coq plug-in re-disambiguates per proof obligation. This results in better proof readability as the variable names are shorter. The main pretty printing is done directly through Coq’s own pretty printing features.

Special attention has also been given when storing the proof obligations to a file. First, the file has a human-readable name, so the user can easily retrieve the proof obligation as well as its proof script. If the lemma is regenerated or reopened, he can step through the proof (and adapt it if necessary), and it does not have to be rewritten from scratch. The different kinds of hypotheses are separated from each other and are given different names. This is in particular important when a proof obligation has been modified (and is considered unproved) by a change to the specification: if the proof did not involve any of the modified hypotheses, it remains valid. This facilitates greatly the reuse of proofs.

One of the key points in interactive verification is the level of difficulty to manipulate the proof assistant in order to construct a proof. To help the user build proof scripts that are both intuitive to read and to make, we have used

the tactic mechanism of Coq [21]. As JACK originally generates proof obligations for automatic verification, numerous hypotheses are added to help the automatic theorem prover. For interactive verification these hypotheses are often useless (and annoying). Therefore we have developed tactics to clean up the proof obligations. There are also tactics⁹ to solve common proof patterns generated by JACK: (*i*) to solve arithmetic goals, (*ii*) to solve proofs by contradiction, (*iii*) to solve array-specific proof obligations, and (*iv*) to solve proof obligations related to assignments. Finally, the Coq plug-in also allows automatic resolution of proof obligations using generic proof scripts, and application-specific tactics can be defined to be used both for interactive verification and with automatic resolution.

7.2 JACK with Coq in Eclipse

An important feature of JACK is that all development can be done inside Eclipse. Therefore, the Coq plug-in contains an editor for Coq, called CoqEditor. CoqEditor provides a way to interact directly with Coq through Eclipse’s Java environment, so the user can process and edit Coq files (containing proof scripts or user-defined tactics). CoqEditor resembles the Isabelle plug-in in Proof General Eclipse [40], but uses a more light-weight approach. It has keyboard shortcuts similar to CoqIde (the current Coq graphical interface, written in OCaml¹⁰). Of course, it provides syntax highlighting and one can interactively process a Coq file. In addition, CoqEditor has an outline view, that summarises the structure of the currently edited Coq file in a tree-like representation (this is especially useful to see the modules hierarchy), and an incremental indexing feature, that allows the user to jump directly from a keyword to its definition.

7.3 Native Specifications

When specifying complex applications, often one needs advanced data structures. It is a major challenge how to specify these in a way that is suitable for verification (see Challenge 1 in [30]). A possible way to do this in JML is by using so-called model classes, but this makes verification awkward, because all operations on these data structures have to be specified by pre-post-condition specifications. A more convenient approach is to use constructs that are specific to the logic of the prover in which the proofs will be developed. This is exactly the functionality provided by the `native` construct [16], *i.e.*, it relates declarations in the JML specification directly to the logic of the underlying prover. We have implemented the `native` construct for Coq, but the same principle can be used to support any other prover.

The `native` construct can be used for types and methods. A `native` method is a specification-only method that has no body and no (JML) specification. It must terminate normally and cannot have any side-effects. A `native` type is a

⁹ See <http://www-sop.inria.fr/everest/soft/Jack/doc/plugin/coq/Prelude/> for a full description of the different tactics.

¹⁰ Available via the Coq distribution (<http://coq.inria.fr>).

In JML we define:

```
/*@ public native class IntList {
    public native IntList append (IntList l);
    public native static IntList create();
    ...
    public native static IntList toList (int [] tab);
} @*/
```

And in Coq:

```
Definition IntList := list t_int.
Definition IntList_create: IntList := nil.
Definition IntList_append: list t_int -> list t_int -> list t_int := app.
...
```

Fig. 7. The definition of the native type `IntList`

```
//@ ghost IntList list;

/*@ requires (tab!=null) && list.equals(IntList.toList(tab))
    assignable tab[0 .. (tab.length -1)], list;
    ensures list.equals(IntList.toList(tab)) &&
        list.isSorted() && list.isInjection(\old(list));
    @*/
public void sort() {if(tab.length > 0) sort(0, tab.length -1);}
```

Fig. 8. Specification of method `sort` with native construct

type to use with specification methods (native methods as well as JML’s `model` methods). Both are related to constructs defined in the proof obligation’s target language: native types are bound to types and native methods to function definitions. When generating the proof obligations, the native constructs are treated as uninterpreted function symbols. The user then specifies in a Coq file how the function symbols are bound to constructs in Coq.

For example, Figure 7 defines a native type `IntList` and binds it to the type of list of integers in Coq. This allows to use Coq’s list library in the proofs. Using the native type declaration, the specification of the `sort` method (from Figure 1 on page 156) can be rewritten as in Figure 8. Notice that this results in more readable and natural annotations, because instead of relying on arrays, we can write it directly in the proof obligation’s target language syntax. The use of the native construct also allows the user to define more easily auxiliary lemmas that can be used to prove the proof obligations and to add automation to proof scripts.

8 Conclusions

This paper describes the main characteristics of JACK, the Java Applet Correctness Kit, a tool set for the validation of security and functional behaviour

properties for Java applications. We have focused in particular on the features that distinguish JACK from other similar tools:

- the integration into a standard IDE;
- a user interface that helps to understand the proof obligations;
- the implementation of an algorithm to generate “obvious” annotations;
- the implementation of an algorithm to encode high-level security properties with JML annotations;
- support for the verification of both source code *and* bytecode; and
- support for interactive verification, both practical (development of user interface and tactics) and theoretical (`native` construct to link annotations with the logic of the underlying theorem prover).

The JACK tool has been used for several small to medium-scale case studies. First of all, we have shown how BML annotations can be used to guarantee resource policies related to memory consumption of bytecode applications [5]. In addition, we have also shown how the verification of exception-freeness at bytecode level can be used to reduce the footprint of Java-to-native compilation schemes. Executable code typically contains run-time checks to decide whether an exception should be thrown. But if it can be proven statically that the exception *never* will be thrown, there is no need for the executable code to contain the run-time checks [20].

Development and maintenance of a verification tool for a realistic programming language is a major effort. During the last decade several such tools have been developed (see the related work section below). This has resulted in a drastic improvement of the technologies available to verify applications. However, we believe that now the moment has come to combine the different technologies, and to bundle this into one powerful verification tool. Development of such a tool is one of the goals of the IST Mobius project. It is foreseen that all technology developed around JACK that distinguish it from other verification tools will be integrated in this single verification tool.

Related work. Several other tools exist aiming at the static verification of JML-annotated Java code, but JACK distinguishes itself from these tools by the features described in this paper. We briefly describe the most relevant other tools. ESC/Java [17] is probably the most used tool. It also aims at a high level of automation, but makes an explicit trade-off between soundness, completeness and automation. The Jive tool [34] uses a Hoare-logic for program verification, requiring much more user interaction. The Key tool [7] uses a dynamic logic approach, where program verification resembles program simulation. The LOOP tool [8] translates both the program and the annotations into specifications in the logic of the PVS theorem prover. Both a Hoare logic and a weakest precondition calculus have been proven sound in PVS, and can be used to verify whether the program respects its specification.

Krakatoa [33] translates JML-annotated programs into an intermediate format, for which the Why tool generates proof obligations. Krakatoa allows the user to specify algebraic specifications as part of the annotations, and use these

in the verification. This resembles the `native` construct, however, the `native` construct directly allows one to use the full expressiveness of the logic of the underlying prover, and to directly reuse any (library) results already proven about the data types. Moreover, the use of the native construct allows one to keep on using a Java-like syntax in the annotations.

The Spec#/Boogie project [3] aims at the specification and verification of annotated C# programs. As JACK, the tool also provides support for verification at source code and bytecode level. However, they do not compile source code specifications into bytecode specifications, that can be shipped with the application. Further, the project mainly aims at automatic verification, and does not provide support for annotation generation.

There are several projects that aim at annotating bytecode: JVer is a tool to verify annotated bytecode [15], but they do not have a special bytecode specification language. The Extended Virtual Platform project aims at developing a framework that allows to compile JML annotations, to allow run-time checking [2], but they do not allow to write specifications directly at bytecode level.

Most approaches to ensuring high-level security properties are based on run-time monitoring, see *e.g.*, [4,38,24,18]. However, run-time monitoring is not an option for trusted personal devices: for the user it would be unacceptable to be blocked in the middle of an application, because of a security violation.

References

1. Abrial, J.-R.: The B Book, Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Alagić, S., Royer, M.: Next generation of virtual platforms. Article in odbms.org (October 2005), http://odbms.org/about_contributors_alagic.html
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 151–171. Springer, Heidelberg (2005)
4. Bartetzko, D., Fischer, C., Möller, M., Wehrheim, H.: Jass – Java with Assertions. In: Havelund, K., Roşu, G. (eds.) ENTCS, vol. 55(2), Elsevier Publishing, Amsterdam (2001)
5. Barthe, G., Pavlova, M., Schneider, G.: Precise analysis of memory consumption using program logics. In: Software Engineering and Formal Methods, pp. 86–95. IEEE Press, Los Alamitos (2005)
6. Barthe, G., Rezk, T., Saabas, A.: Proof obligations preserving compilation. In: Dimitrakos, T., Martinelli, F., Ryan, P.Y.A., Schneider, S. (eds.) FAST 2005. LNCS, vol. 3866, pp. 112–126. Springer, Heidelberg (2006)
7. Beckert, B., Hähle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
8. van den Berg, J., Jacobs, B.: The LOOP compiler for Java and JML. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 299–312. Springer, Heidelberg (2001)
9. Bieber, P., Cazin, J., Girard, P., Lanet, J.-L., Wiels, V., Zanon, G.: Checking secure interactions of smart card applets. Journal of Computer Security 10(4), 369–398 (2002)

10. Breunese, C., Cataño, N., Huisman, M., Jacobs, B.: Formal methods for smart cards: an experience report. *Science of Computer Programming* 55(1-3), 53–80 (2005)
11. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. In: Arts, T., Fokkink, W. (eds.) *Workshop on Formal Methods for Industrial Critical Systems*. Electronic Notes in Theoretical Computer Science, vol. 80, pp. 73–89. Elsevier Science, Inc, Amsterdam (2003) Preprint University of Nijmegen (TR NIII-R0309)
12. Burdy, L., Huisman, M., Pavlova, M.: Preliminary design of BML: A behavioral interface specification language for Java bytecode. In: *Fundamental Approaches to Software Engineering (FASE 2007)*. LNCS, vol. 4422, pp. 215–229. Springer, Heidelberg (2007)
13. Burdy, L., Pavlova, M.: Java bytecode specification and verification. In: *Symposium on Applied Computing*, pp. 1835–1839. Association of Computing Machinery Press (2006)
14. Burdy, L., Requet, A., Lanet, J.-L.: Java applet correctness: A developer-oriented approach. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) *FME 2003*. LNCS, vol. 2805, pp. 422–439. Springer, Heidelberg (2003)
15. Chander, A., Espinosa, D., Islam, N., Lee, P., Nacula, G.: JVer: A Java Verifier. In: Etessami, K., Rajamani, S.K. (eds.) *CAV 2005*. LNCS, vol. 3576, Springer, Heidelberg (2005)
16. Charles, J.: Adding native specifications to JML. In: *Workshop on Formal Techniques for Java Programs* (2006)
17. Cok, D., Kiniry, J.R.: ESC/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) *CASSIS 2004*. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
18. Colcombet, T., Fradet, P.: Enforcing trace properties by program transformation. In: *Principles of Programming Languages, POPL'00*, pp. 54–66. ACM Press, New York (2000)
19. Coq development team: The Coq proof assistant reference manual V8.0. Technical Report 255, INRIA, France (mars 2004), <http://coq.inria.fr/doc/main.html>
20. Courbot, A., Pavlova, M., Grimaud, G., Vandewalle, J.J.: A low-footprint Java-to-native compilation scheme using formal methods. In: Domingo-Ferrer, J., Posegga, J., Schreckling, D. (eds.) *CARDIS 2006*. LNCS, vol. 3928, pp. 329–344. Springer, Heidelberg (2006)
21. Delahaye, D.: A tactic language for the system Coq. In: Parigot, M., Voronkov, A. (eds.) *LPAR 2000*. LNCS (LNAI), vol. 1955, pp. 85–95. Springer, Heidelberg (2000)
22. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. *Journal of the Association of Computing Machinery* 52(3), 365–473 (2005)
23. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM* 18(8), 453–457 (1975)
24. Erlingsson, U.: The Inlined Reference Monitor Approach to Security Policy Enforcement. PhD thesis, Department of Computer Science, Cornell University. Available as Technical Report 2003-1916 (2003)
25. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27(2), 1–25 (2001)
26. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) *FME 2001*. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)

27. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: Generating compact verification conditions. In: Principles of Programming Languages, pp. 193–205. New York, USA. Association of Computing Machinery Press (2001)
28. Jacobs, B.: Weakest precondition reasoning for Java programs with JML annotations. *Journal of Logic and Algebraic Programming* 58, 61–88 (2004)
29. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes* 31, 1–38 (2006)
30. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing* (to appear, 2007)
31. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Kiniry, J.: JML Reference Manual. In: Progress. Department of Computer Science, Iowa State University (July 2005), Available from <http://www.jmlspecs.org>
32. Lindholm, T., Yellin, F.: The Java™ Virtual Machine Specification, 2nd edn. Sun Microsystems, Inc. (1999), <http://java.sun.com/docs/books/vmspec/>
33. Marché, C., Paulin-Mohring, C., Urbain, X.: The Krakatoa tool for certification of Java/JavaCard programs annotated with JML annotations. *Journal of Logic and Algebraic Programming* 58, 89–106 (2004)
34. Meyer, J., Poetzsch-Heffter, A.: An architecture of interactive program provers. In: Graf, S., Schwartzbach, M. (eds.) ETAPS 2000 and TACAS 2000. LNCS, vol. 1785, pp. 63–77. Springer, Heidelberg (2000)
35. Necula, G.C.: Proof-carrying code. In: Principles of Programming Languages, pp. 106–119, New York, USA. Association of Computing Machinery Press (1997)
36. Pavlova, M.: Specification and verification of Java bytecode. PhD thesis, Université de Nice Sophia-Antipolis (2007)
37. Pavlova, M., Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L.: Enforcing high-level security properties for applets. In: Paradin, P., Quisquater, J.-J. (eds.) CARDIS 2004, Kluwer Academic Publishing, Dordrecht (2004)
38. Schneider, F.B.: Enforceable security policies. Technical Report TR99-1759, Cornell University (October 1999)
39. The Coq Development Team: The Coq Proof Assistant Reference Manual – Version V8.1 (July 2006), <http://coq.inria.fr>
40. Winterstein, D., Aspinall, D., Lüth, C.: Proof General/Eclipse: A generic interface for interactive proof. In: International Workshop on User Interfaces for Theorem Provers 2005 (UITP’05) (2005)

Towards a Formal Framework for Computational Trust

(Extended Abstract)

Vladimiro Sassone¹, Karl Krukow², and Mogens Nielsen²

¹ ECS, University of Southampton

² BRICS*, University of Aarhus

Abstract. We define a mathematical measure for the quantitative comparison of probabilistic computational trust systems, and use it to compare a well-known class of algorithms based on the so-called beta model. The main novelty is that our approach is formal, rather than based on experimental simulation.

1 Introduction

Computational trust is an abstraction inspired by the human concept of trust which aims at supporting decision-making by computational agents in the presence of unknown, uncontrollable and possibly harmful entities and in contexts where the lack of reliable information makes classical techniques useless. Such is for instance the case of open networks and ubiquitous computing, where it is entirely unrealistic to assume a priori level of understanding of the environment. Although it would be reductive to think of computational trust as a technique limited to just security, the latter certainly provides an important class of applications where, in general, access to resources is predicated on control policies that depend on the trust relationships in act between their managers and consumers.

As expected of an ineffable idea deeply linked with human emotions and experience, trust appears in computing in several very different forms, from description and specification languages to middleware, from social networks and management of credential to human-computer interaction. These rely in different degrees on a variety of underpinning mathematical theories, including e.g. logics, game theory, semantics, algorithmics, statistics, and probability theory. We focus here on systems where trust in a computational entity is interpreted as the expectation of certain future behaviour based on behavioural patterns of the past, and concern ourselves with the foundations of such probabilistic systems. In particular, we aim at establishing formal probabilistic models for computational trust and their fundamental properties.

In the area of computational trust one common classification distinguishes between ‘probabilistic’ and ‘non-probabilistic’ models (cf. e.g. [1, 2, 3, 11] for the latter and [6, 16, 10, 14] for the former). The non-probabilistic systems vary considerably and need further classification (e.g., as social networks or cognitive); in contrast, the probabilistic systems usually have common objectives and structure: they assume a particular

* BRICS: Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

(probabilistic) model for principal behaviour at the outset, and then put forward algorithms for approximating such behaviour and thus making predictions. In such models the trust information about a principal is typically information about its past behaviour, its history. Histories do not immediately classify principals as ‘trustworthy’ or ‘untrustworthy,’ as ‘good’ or ‘bad;’ rather, they are used to estimate the probability of potential outcomes arising in a next interaction with an entity. Probabilistic models (called ‘game-theoretical’ by Sabater and Sierra [16]) rely on Gambetta’s view of trust [7]:

“... *trust is a particular level of the subjective probability with which an agent assesses that another agent or group of agents will perform a particular action, both before he can monitor such action (or independently of his capacity ever to be able to monitor it) and in a context in which it affects his own action.*”

The contribution of this paper is inspired by such a predictive view of trust, and follows the Bayesian approach to probability theory as advocated in e.g. [8] and exploited in works such as [13, 6, 17]. In particular, we borrow ideas from information theory to measure the quality of the behaviour-approximation algorithms and, therefore, suggest a formal framework for the comparison of probabilistic models.

Bayesian analysis consists of formulating hypotheses on real-world phenomena of interest, running experiments to test such hypotheses, and thereafter updating the hypotheses –if necessary– to provide a better explanation of the experimental observations, a better fit of the hypotheses to the observed behaviours. By formulating it in terms of conditional probabilities on the space of interest, this procedure is expressed succinctly in formulae by Bayes’ Theorem:

$$Prob(\Theta | X) \propto Prob(X | \Theta) \cdot Prob(\Theta).$$

Reading from left to right, the formula is interpreted as saying: the probability of the hypotheses Θ posterior to the outcome of experiment X is proportional to the likelihood of such outcome under the hypotheses multiplied by the probability of the hypotheses prior to the experiment.¹ In the present context, the prior Θ will be an estimate of the probability of each potential outcome in our next interaction with principal p , whilst the posterior will be our amended estimate after one such interaction took place with outcome X .

It is important to observe here that $Prob(\Theta | X)$ is in a sense a second order notion, and we are not interested in computing it for any particular value of Θ . Indeed, as Θ is the unknown in our problem, we are interested in deriving the entire distribution in order to compute its expected value, and use it as our next estimate for Θ .

In order to make this discussion concrete, let us focus on a model of binary outcomes, which is very often used in practice. Here Θ can be represented by a single probability θ_p , the probability that principal p will behave benevolently, i.e., that an interaction with p will be successful. In this case, a sequence of n experiments $X = X_1 \cdots X_n$ is a sequence of binomial (Bernoulli) trials, and is modelled by a binomial distribution

$$Prob(X \text{ consists of } k \text{ successes}) = \theta_p^k (1 - \theta_p)^{n-k}.$$

¹ We shall often omit the proportionality factor, as that is uniquely determined as the constant that makes the right-hand side term a probability distribution. In fact, it equals $Prob(X)^{-1}$.

It turns out that if the prior Θ follows a β -distribution, say $B(\alpha, \beta) \propto \Theta_p^{\alpha-1} (1 - \Theta_p)^{\beta-1}$ of parameters α and β , then so does the posterior: viz., if X is an n -sequence of k successes, $Prob(\Theta | X)$ is $B(\alpha + k, \beta + n - k)$, the β -distribution of parameters $\alpha + k$ and $\beta + n - k$. This is a particularly happy circumstance when it comes to apply Bayes' Theorem, because it makes it straightforward to compute the posterior distribution and its expected value from the prior and the observations; it is known in the literature as the condition that the β -distribution family is a *conjugate prior* for the binomial trials.

In [14] we extend the framework from events with binary (success/failure) outcomes to complex, structured outcomes: namely, the configurations of finite, confusion-free event structures. In the new framework, our Bayesian analysis relies on observing sequences of event structure configurations –one event at the time– to ‘learn’ (i.e., estimate) the probability of each configuration occurring as the outcome of the next complex (sequence of elementary) interactions.

In this paper we illustrate our main technical results from [14], viz., the definition of a formal measure expressing the quality of probabilistic computational trust systems in various application environments. The measure is based on the so-called Kullback-Leibler divergence [12], also known as *information divergence* or *relative entropy*, used in the information theory literature to measure the ‘distance’ from an approximation to a known target probability distribution. Here we adapt it to measure how well an computational trust algorithm approximates the ‘true’ probabilistic behaviours of computing entities and, therefore, to provide a formal benchmark for the comparison of such algorithms. As an illustration of the applicability of the theory, we present theoretical results within the field, regarding a whole class of existing probabilistic trust algorithms. To our knowledge, no such approach has been proposed previously (but cf. [4] for an application of similar concepts to anonymity), and these presents the first formal results ever in way of comparison of computational trust algorithms.

Structure of the paper. The paper is organised as follows. In §2 we make precise the scenario illustrated informally in the Introduction, while in §3 we illustrate our results on the formal of computational trust algorithms. We remand the reader to [14] for the formal proofs. Finally, §4 reflects on some of the basic hypotheses of the probabilistic models considered in the paper, and points forward to future research aimed at relaxing them.

2 Bayesian Models for Trust

At the outset, Bayesian trust models are based on the assumption that principals behave in a way that can profitably be approximated by fixed probabilities. Accordingly, while interacting with principal p one will constantly experience outcomes as following an immutable probability distribution Θ_p . Such assumption may of course be unrealistic in several real-world scenarios, and we shall discuss in §4 a research programme aimed to lift it; for the moment however, we proceed to explore where such an assumption leads us.

Our overall goal is to obtain an estimate of Θ_p in order to inform our future policy of interaction with p . Computational trust algorithms attempt to do this using Bayesian

analysis on the history of past interactions with p . Let us fix a probabilistic model of principal behaviour, that is a set of basic assumptions on the way principals behave, say λ , and then consider the behaviour of a single, fixed principal p . We shall focus on algorithms for the following problem: let X be an interaction history $x_1 x_2 \dots x_n$ obtained by interacting n times with p and observing in sequence outcomes x_i out of a set $\{y_1, \dots, y_k\}$ of possible outcomes. A probabilistic computational trust algorithm, say \mathcal{A} , outputs on input X a probability distribution $\mathcal{A}(\cdot | X)$ on the outcomes $\{y_1, \dots, y_k\}$. That is, \mathcal{A} satisfies:

$$\mathcal{A}(y_i | X) \in [0, 1] \quad (i = 1, \dots, k) \qquad \sum_{i=1}^k \mathcal{A}(y_i | X) = 1.$$

Such distribution is meant to approximate a θ_p under the hypotheses λ . To make this precise, let us assume that the probabilistic model λ defines the following probabilities:

- $Prob(y_i | X \lambda)$: the probability of “observing y_i in the next interaction in the model λ , given the past history X ,”
- $Prob(X | \lambda)$: the *a priori* probability of “observing X in the model λ .”

Now, $Prob(\cdot | X \lambda)$ defines the ‘true’ distribution on outcomes for the next interaction (according to the model); in contrast, $\mathcal{A}(\cdot | X)$ aims at approximating it. We shall now propose a generic measure to ‘score’ specific algorithms \mathcal{A} against given probability distributions. The score, based on the so-called Kullback-Leibler divergence, is a measure of how well the algorithm approximates the ‘true’ probabilistic behaviour of principals.

3 Towards Comparing Probabilistic Trust-Based Systems

Closely related to Shannon’s notion of entropy, Kullback and Leibler’s information divergence [12] is a measure of the distance between two probability distributions. For $p = (p_1, \dots, p_k)$ and $q = (q_1, \dots, q_k)$ distributions on a set of k events, the Kullback-Leibler divergence from p to q is defined by

$$D_{KL}(p \parallel q) = \sum_{i=1}^k p_i \log_2(p_i/q_i).$$

Information divergence resembles a distance in the mathematical sense: it can be proved that D_{KL} satisfies $D_{KL}(p \parallel q) \geq 0$, where the equality holds if and only if $p = q$; however, D_{KL} fails to be symmetric. We adapt D_{KL} to score the distance between algorithms by taking the its average over possible input sequences, as illustrated below.

For each $n \in \mathbb{N}$, let \mathbf{O}^n denote the set of interaction histories $X_1 \dots X_n$ of length n . Define D_{KL}^n , the *n*th expected Kullback-Leibler divergence from λ to \mathcal{A} as:

$$D_{KL}^n(\lambda \parallel \mathcal{A}) = \sum_{X \in \mathbf{O}^n} Prob(X | \lambda) \cdot D_{KL}(Prob(\cdot | X \lambda) \parallel \mathcal{A}(\cdot | X)),$$

Note that, for each possible input sequence $X \in \mathcal{O}^n$, we evaluate the algorithm's performance as $D_{\text{KL}}(\text{Prob}(\cdot | X \lambda) \| \mathcal{A}(\cdot | X))$, i.e. we accept that some algorithms may perform poorly on very unlikely training sequences X , whilst providing excellent results frequent inputs. Hence, we weigh the performance on each input X by the intrinsic probability of sequence X . In other terms, we compute the *expected* information divergence for inputs of size n .

While Kullback and Leibler's information divergence is a well-established measure in statistics, to our knowledge measuring probabilistic algorithms via D_{KL}^n is new. Due to the relation to Shannon's information theory, one can interpret $D_{\text{KL}}^n(\lambda \| \mathcal{A})$ quantitatively as the expected number of *bits of information* one would gain by knowing the 'true' distribution $\text{Prob}(\cdot | X \lambda)$ on all training sequences of length n , rather than its approximation $\mathcal{A}(\cdot | X)$.

An example. In order to exemplify our measure, we compare the β -based algorithm of Mui *et al* [13] with the maximum-likelihood algorithm of Aberer and Despotovic [5]. The comparison is possible as the algorithms share the same fundamental assumptions that:

each principal's behaviour is so that there is a fixed parameter θ that at each interaction we have, *independently of anything we know about other interactions*, probability θ of 'success' and, therefore, probability $1 - \theta$ of 'failure.'

We refer to these as the β -model $\lambda_{\mathbf{B}}$. With s and f standing respectively for 'success' and 'failure,' an n -fold experiment is a sequence $X \in \{s, f\}^n$, for some $n > 0$. The likelihood of $X \in \{s, f\}^n$ is given by

$$\text{Prob}(X | \theta \lambda_{\mathbf{B}}) = \theta^{\#_s(X)}(1 - \theta)^{\#_f(X)},$$

where $\#_x(X)$ denotes the number of occurrences x in X . Using \mathcal{A} and \mathcal{B} to denote respectively the algorithm of Mui *et al*, and of Aberer and Despotovic, we have that:

$$\begin{aligned} \mathcal{A}(s | X) &= \frac{\#_s(X) + 1}{n + 2} & \text{and} & & \mathcal{A}(f | X) &= \frac{\#_f(X) + 1}{n + 2}, \\ \mathcal{B}(s | X) &= \frac{\#_s(X)}{n} & \text{and} & & \mathcal{B}(f | X) &= \frac{\#_f(X)}{n}. \end{aligned}$$

For each choice of $\theta \in [0, 1]$ and each choice of training-sequence length n , we can compare the two algorithms by computing and comparing $D_{\text{KL}}^n(\theta \lambda_{\mathbf{B}} \| \mathcal{A})$ and $D_{\text{KL}}^n(\theta \lambda_{\mathbf{B}} \| \mathcal{B})$.

Theorem 1. *If $\theta = 0$ or $\theta = 1$, Aberer and Despotovic's algorithm \mathcal{B} from [5] computes a better approximation of the principal's behaviour than Mui *et al*'s algorithm \mathcal{A} from [13]. In fact, under the assumptions, \mathcal{B} always computes the exact probability of success on any possible training sequence.*

The proof follows easily after observing that under the hypothesis on θ there is only one n -sequence with non-zero probability, viz., either f^n or s^n .

Concerning the same comparison when $0 < \theta < 1$, it suffices to observe that \mathcal{B} assigns probability 0 to s on input f^k for all $k \geq 1$; this results in $D_{\text{KL}}^n(\theta \lambda_{\mathbf{B}} \parallel \mathcal{B}) = \infty$. It follows that \mathcal{A} provides a better approximation.

In order to explore the space of β -based algorithms further, we define a parametric algorithm \mathcal{A}_ϵ , for $\epsilon \geq 0$, that encompasses both \mathcal{A} and \mathcal{B} :

$$\mathcal{A}_\epsilon(s \mid h) = \frac{\#_s(h) + \epsilon}{|h| + 2\epsilon} \quad \text{and} \quad \mathcal{A}_\epsilon(s \mid X) = \frac{\#_f(h) + \epsilon}{|h| + 2\epsilon}.$$

Observe that $\mathcal{A}_0 = \mathcal{B}$ and $\mathcal{A}_1 = \mathcal{A}$.

Let us now study the expression $D_{\text{KL}}^n(\theta \lambda_{\mathbf{B}} \parallel \mathcal{A}_\epsilon)$ as a function of ϵ . It turns out that for each $\theta \neq 1/2$ and independently of n there is a unique $\bar{\epsilon}$ which minimises the distance $D_{\text{KL}}^n(\theta \lambda_{\mathbf{B}} \parallel \mathcal{A}_\epsilon)$. Furthermore, $D_{\text{KL}}^n(\theta \lambda_{\mathbf{B}} \parallel \mathcal{A}_\epsilon)$ is decreasing on the interval $(0, \bar{\epsilon}]$ and increasing on the interval $[\bar{\epsilon}, \infty)$. (Note of course that $D_{\text{KL}}^n(\theta \lambda_{\mathbf{B}} \parallel \mathcal{A}_\epsilon) \rightarrow \infty$ when $\epsilon \rightarrow 0$.) By definition, we have:

$$D_{\text{KL}}^n(\theta \lambda_{\mathbf{B}} \parallel \mathcal{A}_\epsilon) = \sum_{i=0}^n \binom{n}{i} \theta^i (1 - \theta)^{n-i} \left[\theta \log \frac{\theta(n + 2\epsilon)}{i + \epsilon} + (1 - \theta) \log \frac{(1 - \theta)(n + 2\epsilon)}{n - i + \epsilon} \right].$$

By differentiating $D_{\text{KL}}^n(\theta \lambda_{\mathbf{B}} \parallel \mathcal{A}_\epsilon)$ with respect to epsilon, we obtain

$$\frac{d}{d\epsilon} D_{\text{KL}}^n(\theta \lambda_{\mathbf{B}} \parallel \mathcal{A}_\epsilon) = \frac{2\alpha}{n + 2\epsilon} - \sum_{i=0}^n \binom{n}{i} \theta^i (1 - \theta)^{n-i} \left[\frac{\theta\alpha}{i + \epsilon} + \frac{(1 - \theta)\alpha}{n - i + \epsilon} \right],$$

where $\alpha = \log e$ is a positive constant obtained when differentiating the function \log . It is proved in [14] that ϵ nullifies the derivative $dD_{\text{KL}}^n(\theta \lambda_{\mathbf{B}} \parallel \mathcal{A}_\epsilon)/d\epsilon$ if and only if

$$\theta \neq 1/2 \quad \text{and} \quad \epsilon = \frac{2\theta(1 - \theta)}{(2\theta - 1)^2}.$$

In addition to that, one can prove that in fact

$$\frac{d}{d\epsilon} D_{\text{KL}}^n(\theta \lambda_{\mathbf{B}} \parallel \mathcal{A}_\epsilon) < 0 \quad \text{iff} \quad \epsilon < \frac{2\theta(1 - \theta)}{(2\theta - 1)^2}$$

and

$$\frac{d}{d\epsilon} D_{\text{KL}}^n(\theta \lambda_{\mathbf{B}} \parallel \mathcal{A}_\epsilon) > 0 \quad \text{iff} \quad \epsilon > \frac{2\theta(1 - \theta)}{(2\theta - 1)^2}$$

Remarkably, these formulae are independent of n . We have thus the following result.

Theorem 2. *For any $\theta \in [0, 1/2) \cup (1/2, 1]$ there exists $\bar{\epsilon} \in [0, \infty)$ that minimises $D_{\text{KL}}^n(\theta \lambda_{\mathbf{B}} \parallel \mathcal{A}_\epsilon)$ simultaneously for all n ; viz., $\bar{\epsilon} = 2\theta(1 - \theta)/(2\theta - 1)^2$.*

Furthermore, $D_{\text{KL}}^n(\theta \lambda_{\mathbf{B}} \parallel \mathcal{A}_\epsilon)$ is a decreasing function of ϵ in the interval $(0, \bar{\epsilon})$ and increasing in $(\bar{\epsilon}, \infty)$.

This means that unless the principal’s behaviour is completely unbiased, then there exists a unique best $\mathcal{A}_{\bar{\epsilon}}$ algorithm that outperforms all the others, for all n . If instead

$\theta = 1/2$, then the larger the $\bar{\epsilon}$, the better the algorithm. In fact, $\bar{\epsilon}$ tends to ∞ as θ tends to $1/2$. Regarding \mathcal{A} and \mathcal{B} , an application of Theorem 2 tells us that the former is optimal for $\theta = 1/2 \pm 1/\sqrt{12}$, whilst –as anticipated by Theorem 1– the latter is such for $\theta = 0$ and $\theta = 1$.

We remark here that it is not so much the comparison of algorithms \mathcal{A} and \mathcal{B} that interests us; rather, the message is that using formal probabilistic models enables such mathematical comparisons and, more in general, to investigate properties of models and algorithms.

4 Towards a Formal Model of Dynamic Behaviour

Our main motivation for this investigation is to put on formal grounds what we have been seeing in the literature, with the ultimate aim to exploit a sharpened understanding on systems and models. In our view, we succeeded in this to a comforting extent, by presenting the first ever formal framework for the comparisons of computational trust algorithms.

However, our probabilistic models must become more realistic. For example, the β -model of principal behaviour (which we consider to be state-of-the-art) assumes that for each principal p there is a single fixed parameter θ_p so at each interaction, independently of anything else we know, the probability of a ‘good’ outcome is θ_p of the one of ‘bad’ outcome is $1 - \theta_p$. One might argue that this is unrealistic for several applications. In particular, the model allows for no dynamic behaviour, while in reality not only the p is likely to change its behaviour in time, as its environmental conditions change, but p ’s behaviour in interactions with q is likely to depend on q ’s behaviour in interactions with p .

Some beta-based reputation systems attempt to deal with the first problem by introducing so-called ‘forgetting factors.’ Essentially this amounts to choosing a factor $0 \leq \delta \leq 1$, and then each time the parameters (α, β) of the pdf for θ_p are updated, they are also scaled by δ . In particular, when observing a single ‘good’ interaction, (α, β) becomes $(\alpha\delta + 1, \beta\delta)$ rather than (α, β) . Effectively, this performs a form of exponential ‘decay’ on parameters. The idea is that information about old interactions is less relevant than new information, as it is more likely to be outdated. This approach represents a departure from the probabilistic beta model, where all interactions ‘weigh’ equally, and in the absence of any mathematical explanation it is not clear what the exact benefits of this bias towards newer information is. Regarding the second problem, to our knowledge it has not yet been considered in the literature.

Let us point out some ideas towards refining such hypothesis embracing the fact that the behaviour of p depends on its internal state, which is likely to change over time. Suppose we model p as a kind of Markov chain, a probabilistic finite-state system with n states $S = \{1, 2, \dots, n\}$ and n^2 transition probabilities $t_{ij} \in [0, 1]$, with $\sum_{j=1}^n t_{ij} = 1$. After each interaction, p changes state according to t : it takes a transition from state i to state j with probability t_{ij} . Such state-changes are likely in our context to be unobservable: a principal q does not know for certain which state principal p is in. All that q can observe, now as before, is the outcome of its interactions with p ; based on that, it must make inferences on p ’s likely state and future actions. If we accept the finite state assumption

and the Markovian transition probabilities, we can then incorporate unobservable states in the model by using so-called Hidden Markov Models [15].

A discrete *Hidden Markov Model* (HMM) is a tuple $\lambda = (S, \pi, t, O, s)$ where S is a finite set of *states*; π is a distribution on S , the *initial distribution*; $t : S \times S \rightarrow [0, 1]$ is the *transition matrix*, with $\sum_{j \in S} t_{ij} = 1$; finite set O is the set of possible *observations*; and where $s : S \times O \rightarrow [0, 1]$, the *signal*, assigns to each state $j \in S$, a distribution s_j on observations, i.e., $\sum_{o \in O} s_j(o) = 1$.

An example. Consider the HMM in Figure 1. This models a simple two-state process with two possible observable outputs a and b . For example, this could model a channel which can forward a packet or drop it. State 1 models the normal mode of operation, whereas state 2 models operation under high load. Suppose that output a means ‘packet forwarded’ and output b means ‘packet dropped.’ Most of the time, the channel is in state 1, and packets are forwarded with probability .95; occasionally the channel will transit to state 2 where packets are dropped with probability .95. Although this example is just meant to illustrate a simple HMM, we expect that by tuning their parameters Hidden Markov Models can provide an interesting model many of the dynamic behaviours needed for probabilistic trust-based systems.

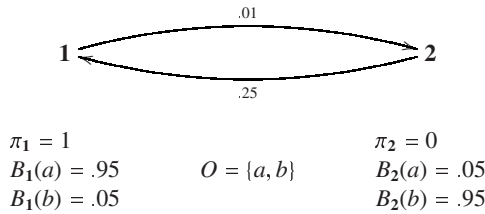


Fig. 1. Example Hidden Markov Model

Consider now an observation sequence, $h = a^{10}b^2$ (that is ten a 's followed by two b 's), which is reasonably probable in our model on Figure 1. The final fragment consisting of two consecutive occurrences of b 's makes it likely that a state-change from 1 to 2 has occurred. Nevertheless, a simple counting algorithm, say \mathcal{H} , would probably assign high probability to the event that a will happen next:

$$\mathcal{H}(a | h) = \frac{\#_a(a^{10}b^2) + 1}{|h| + 2} = 11/14 \sim .80$$

However, if a state-change has indeed occurred, that probability would be as low as .05.

Suppose now exponential decay is used, e.g., as in the Beta reputation system [9], with a factor of $\delta = .5$. This means that the last observation weighs approximately the same as the rest of the history; in such a case, the algorithm would adapt quickly, and assign probability $\mathcal{H}(a | h) \sim .25$, which is a much better estimate. However, suppose that we now observe bb and then another a . Again this would be reasonably likely in state 2, and would make a state-change to 1 probable in the model. The exponential

forgetting would assign a high weight to a , but also a high weight to b , because the last four observations were b 's. In a sense, perhaps the algorithm adapts 'too quickly,' it is too sensitive to new observations. So, no matter what δ is, it appears easy to describe situations where it does not reach its intended objective; our main point here is the same as for our comparisons of computational trust algorithms in §3: that the underlying assumptions behind a computational idea (e.g., the exponential decay) need to be specified, and that formal models for principal's behaviour (e.g., HMMs) may serve the purpose, allowing precise questions on the applicability of the computational idea.

5 Conclusion

Our 'position' on computational trust research is that any proposed system should be able to answer two fundamental questions precisely: What are the assumptions about the intended environments for the system? And what is the objective of the system? An advantage of formal probabilistic models is that they enable rigorous answers to these questions.

Among the several benefits of formal probabilistic models, we have focussed on the possibility to compare algorithms, say \mathcal{X} and \mathcal{Y} , that work under the same assumption on principal behaviours. The comparison technique we proposed relies on Kullback and Liebler's information diverge, and consists of measuring which algorithm best approximates the 'true' principal behaviour postulated by the model. For example, in order to compare \mathcal{X} and \mathcal{Y} in the model λ , we propose to compute and compare

$$D_{\text{KL}}^n(\lambda \parallel \mathcal{X}) \quad \text{and} \quad D_{\text{KL}}^n(\lambda \parallel \mathcal{Y}).$$

Note that no simulations of algorithms \mathcal{X} and \mathcal{Y} are necessary; the mathematics provide a theoretical justification –rooted in concepts from Information Theory– stating e.g. that “in environment λ , on average, algorithm \mathcal{X} outperforms algorithm \mathcal{Y} on training sequences of length n .” Using our method we have successfully in shown a theoretical comparison between two β -based algorithms well-known in the literature. Moreover, we explored the entire space of β -based algorithms and illustrated constructively that for each principal behaviour θ , there exists a best approximating algorithm. Remarkably, this does not depend on n , the length of the training sequence. More generally, another type of property one might desire to prove using the notion of information diverge is that $\lim_{n \rightarrow \infty} D_{\text{KL}}^n(\lambda \parallel \mathcal{X}) = 0$, meaning that algorithm \mathcal{X} approximates the true principal behaviour to an arbitrary precision, given a sufficiently long training sequence.

References

1. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.D.: The role of trust management in distributed systems security. In: Vitek, J. (ed.) *Secure Internet Programming*. LNCS, vol. 1603, pp. 185–210. Springer, Heidelberg (1999)
2. Carbone, M., Nielsen, M., Sassone, V.: A formal model for trust in dynamic networks. In: *Proceedings from Software Engineering and Formal Methods (SEFM'03)*, IEEE Computer Society Press, Los Alamitos (2003)

3. Carbone, M., Nielsen, M., Sassone, V.: A calculus for trust management. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 161–173. Springer, Heidelberg (2004)
4. Chatzikokolakis, K., Palamidessi, C., Panangaden, P.: Anonymity protocols as noisy channels. In: Proceedings of TGC'06. LNCS, Springer, Heidelberg (to appear, 2007)
5. Despotovic, Z., Aberer, K.: A probabilistic approach to predict peers' performance in P2P networks. In: Klusch, M., Ossowski, S., Kashyap, V., Unland, R. (eds.) CIA 2004. LNCS (LNAI), vol. 3191, pp. 62–76. Springer, Heidelberg (2004)
6. Despotovic, Z., Aberer, K.: P2P reputation management: Probabilistic estimation vs. social networks. *Computer Networks* 50(4), 485–500 (2006)
7. Gambetta, D.: Can we trust trust. In: Gambetta, D. (ed.) *Trust: Making and Breaking Co-operative Relations*, pp. 213–237. University of Oxford, Department of Sociology, Ch. 13. Electronic edition (2000),
<http://www.sociology.ox.ac.uk/papers/gambetta213-237.pdf>
8. Jaynes, E.T.: *Probability Theory: The Logic of Science*. Cambridge University Press, Cambridge (2003)
9. Jøsang, A., Ismail, R.: The beta reputation system. In: Proceedings from the 15th Bled Conference on Electronic Commerce, Bled (2002)
10. Krukow, K.: *Towards a Theory of Trust for the Global Ubiquitous Computer*. PhD thesis, University of Aarhus, Denmark (August 2006), Available at <http://www.brics.dk/~krukow>
11. Krukow, K., Nielsen, M., Sassone, V.: A logical framework for reputation systems. *Journal of Computer Security* (to appear, 2007), Available online <http://eprints.ecs.soton.ac.uk/13656/>
12. Kullback, S., Leibler, R.A.: On information and sufficiency. *Annals of Mathematical Statistics* 22(1), 79–86 (1951)
13. Mui, L., Mohtashemi, M., Halberstadt, A.: A computational model of trust and reputation (for ebusinesses). In: Proceedings from 5th Annual Hawaii International Conference on System Sciences (HICSS'02), p. 188. IEEE, Los Alamitos (2002)
14. Nielsen, M., Krukow, K., Sassone, V.: A bayesian model for event-based trust. In: *Festschrift for Gordon D. Plotkin*. ENTCS, Elsevier, Amsterdam (to appear, 2007)
15. Rabiner, L.R.: A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE* 77(2), 257–286 (1989)
16. Sabater, J., Sierra, C.: Review on computational trust and reputation models. *Artificial Intelligence Review* 24(1), 33–60 (2005)
17. Teacy, W.T.L., Patel, J., Jennings, N.R., Luck, M.: Coping with inaccurate reputation sources: experimental analysis of a probabilistic trust model. In: *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pp. 997–1004. ACM Press, New York (2005)

On Recursion, Replication and Scope Mechanisms in Process Calculi

Jesús Aranda¹, Cinzia Di Giusto², Catuscia Palamidessi³,
and Frank D. Valencia⁴

¹ Universidad del Valle Cali, Colombia and LIX École Polytechnique
Paris*, France

`jesus.aranda@lix.polytechnique.fr`

² Dip. Scienze dell'Informazione, Università di Bologna
Bologna, Italy

`digiusto@cs.unibo.it`

³ INRIA-LIX, École Polytechnique
Paris, France

`catuscia@lix.polytechnique.fr`

⁴ CNRS - LIX École Polytechnique
Paris, France

`frank.valencia@lix.polytechnique.fr`

Abstract. In this paper we shall survey and discuss in detail the work on the relative expressiveness of recursion and replication in various process calculi. Namely, CCS, the π -calculus, the Ambient calculus, Concurrent Constraint Programming and calculi for Cryptographic Protocols. We shall give evidence that the ability of expressing recursive behaviour via replication often depends on the scoping mechanisms of the given calculus which compensate for the restriction of replication.

1 Introduction

Process calculi such as CCS [Mil89], the π -calculus [Mil99] and Ambients [CG98] are among the most influential formal methods for modelling and analyzing the behaviour of concurrent systems; i.e. systems consisting of multiple computing agents, usually called *processes*, that interact with each other. A common feature of these calculi is that they treat processes much like the λ -calculus treats computable functions. They provide a language in which the structure of *terms* represents the structure of processes together with an *operational semantics* to represent computational steps. Another common feature, also in the spirit of the λ -calculus, is that they pay special attention to economy. That is, there are few process constructors, each one with a distinct and fundamental role.

* The work of Jesús Aranda has been supported by COLCIENCIAS (Instituto Colombiano para el Desarrollo de la Ciencia y la Tecnología "Francisco José de Caldas") and ÉGIDE (Centre français pour l'accueil et les échanges internationaux).

For example, a typical process term is the *parallel composition* $P \mid Q$, which is built from the terms P and Q with the constructor \mid and it represents the process that results from the parallel execution of the processes P and Q . Another typical term is the *restriction* $(\nu x)P$ which represents a process P with a private resource x —e.g., a location, a link, or a name. An operational semantics may dictate that if P can reduce to (or evolve into) P' , written $P \longrightarrow P'$, then we can also have the reductions $P \mid Q \longrightarrow P' \mid Q$ and $(\nu x)P \longrightarrow (\nu x)P'$.

Infinite behaviour is ubiquitous in concurrent systems (e.g., browsers, search engines, reservation systems). Hence, it ought to be represented by process terms. Two standard term representations of them are *recursive process expressions* and *replication*.

Recursive process expressions are reminiscent of the recursive expressions used in other areas of computer science, such as for example Functional Programming. They may come in the form $\mu X.P$ where P may have occurrences of X . The process $\mu X.P$ behaves as P with the (free) occurrences of X replaced by $\mu X.P$. Another presentation of recursion is by using *parametric processes* of the form $A(y_1, \dots, y_n)$ each assumed to have a unique, possibly recursive, *definition* $A(x_1, \dots, x_n) \stackrel{\text{def}}{=} P$ where the x_i 's are pairwise distinct, and the intuition is that $A(y_1, \dots, y_n)$ behaves as its P with each y_i replacing x_i .

Replication, syntactically simpler than recursion, takes the form $!P$ and it is reminiscent of Girard's bang operator; an operator used to express unlimited number of copies of a given resource in linear-logic [Gir87]. Intuitively, $!P$ means $P \mid P \mid \dots$; an unbounded number of copies of the process P .

Now, it is not uncommon that a given process calculus, originally presented with one form of defining infinite behavior, is later presented with the other. For example, the π -calculus was originally presented with recursive expressions and later with replication [MPW92]. The Ambient calculus was originally presented with replication and later with recursion [LS03]. This is reasonable as a variant may simplify the presentation of the calculus or be tailored to specific applications.

From the above intuitive description it should be easy to see that $\mu X.(P \mid X)$ expresses the unbounded parallel behaviour of $!P$. It is less clear, however, whether replication can be used to express the unbounded behaviour of $\mu X.P$. In particular, processes that allows for unboundedly many *nested* restrictions as, for example, in $\mu X.(\nu x)(P \mid X)$ which behaves as $(\nu x)(P \mid (\nu x)(P \mid (\nu x)(P \mid \dots)))$. In fact, the ability of expressing recursive behaviours via replication depends on the particular process calculus under consideration. We shall see that typically that scoping mechanisms such as restriction (or hiding) and name passing play a key role in the recursion vs replication expressiveness question.

The above discussion raises the issue of *expressiveness*. What does it mean for one variant to be as expressive as another? The answer to this question is definite in the realm of computability theory via the notion of language equivalence. In concurrency theory, however, this issue is not quite settled.

One approach to comparing expressiveness of two given process calculus variants is by comparing them w.r.t. some standard process equivalence, say \sim . If

for every process P in one variant there is a Q in the other variant such that $Q \sim P$ then we say that the latter variant is at least as expressive as the former.

Another approach consists in telling two variants apart by showing that in one variant one can solve some fundamental problem (e.g., leader election) while in the other one cannot. It should be noticed that, unlike computability theory, the capability of two variants of simulating Turing Machines does not imply equality in their expressiveness. For example, [Pal97] shows that under some reasonable assumptions the asynchronous version of the π -calculus, which can certainly encode Turing Machines, is strictly less expressive than the original calculus.

In this paper, we shall survey and discuss the work on the relative expressiveness of recursion and replication in various process calculi. In particular, CCS, the π -calculus, and the Ambient calculus. We shall begin with the π -calculus, then CCS and then the Ambients calculus. For the simplicity of the presentation we shall consider the polyadic variant of the π -calculus [Mil93]. Finally, we shall also overview the work on this subject in related calculi such as tcc [SJG94] and calculi for Cryptographic Protocols [HS05]. This paper is the extended and revised version of the survey in [PV05].

2 The Polyadic Pi Calculus: $p\pi$

One of the earliest discussions about the relative expressiveness between replication and recursion was in the context of the polyadic π -calculus [Mil93]; one of the main calculi for mobility. It turns out that in this calculus replication is just as expressive as recursion. This result is rather surprising since replication seems such an elementary construct without much control power.

In what follows we shall introduce the polyadic π -calculus and the variants relevant for this paper. The various CCS and Ambients variants will be presented in the next sections as extension/restrictions of the polyadic π -calculus.

2.1 Finite Pi-Calculus

Names are the most primitive entities in the π -calculus. We presuppose a countable set of (port, links or channel) *names*, ranged over by x, y, \dots . For each name x , we assume a *co-name* \bar{x} thought of as *complementary*, so we decree that $\bar{\bar{x}} = x$. We shall use l, l', \dots to range over names and co-names. We use \mathbf{x} to denote a finite sequence of names $x_1 x_2 \cdots x_n$. The other entity in the π -calculus is a *process*. Process are built from names by the following syntax:

$$\begin{aligned}
 P, Q, \dots := & \sum_{i \in I} \alpha_i.P_i \quad | \quad (\nu x)P \quad | \quad P \mid Q & (1) \\
 \alpha := & \bar{x}\mathbf{y} \quad | \quad x(\mathbf{y})
 \end{aligned}$$

where I is a finite set of indexes.

Let us recall briefly some notions as well as the intuitive behaviour of the various constructs.

The construct $\sum_{i \in I} \alpha_i.P_i$ represents a process able to perform one—but only one—of its α_i 's actions and then behave as the corresponding P_i . The actions prefixing the P_i 's can be of two forms: An output $\bar{x}\mathbf{y}$ and an input $x(\mathbf{y})$. In both cases x is called the *subject* and \mathbf{y} the *object*. The action $\bar{x}\mathbf{y}$ represents the capability of sending the names \mathbf{y} on channel x . The action $x(\mathbf{y})$, with no name occurring twice in \mathbf{y} , represents the capability of receiving the names on channel x , say \mathbf{z} , and replacing each y_i with z_i in its corresponding continuation.

Furthermore, in $x(\mathbf{y}).P$ the input actions binds the names \mathbf{y} in P . The other name binder is the *restriction* $(\nu x)P$ which declares a name x private to P , hence bound in P . Given Q we define in the standard way its *bound names* $bn(Q)$ as the set of variables with a bound occurrence in Q , and its *free names* $fn(Q)$ as the set of variables with a non-bound occurrence in Q .

Finally, the process $P \mid Q$ denotes *parallel composition*; P and Q running in parallel.

Convention 1. We write the summation as 0 if $|I| = 0$, and drop the “ $\sum_{i \in I}$ ” if $|I| = 1$. Also we write $\alpha_1.P_1 + \dots + \alpha_n.P_n$ for $\sum_{i \in \{1, \dots, n\}} \alpha_i.P_i$.

For simplicity, we omit “ $()$ ” in processes of the form $x().P$ as well as the “ $.0$ ” in processes of the form $x(\mathbf{y}).0$. We use $(\nu x_1 x_2 \dots x_n)P$ as an abbreviation $(\nu x_1)(\nu x_2) \dots (\nu x_n)P$ and $\prod_{i \in I} P_i$, where $I = \{i_1, \dots, i_n\}$, as an abbreviation of $P_{i_1} \mid \dots \mid P_{i_n}$. Furthermore, $P\sigma$, where $\sigma = \{z_1/y_1, \dots, z_n/y_n\}$, denotes the process that results from the substitution in P of each z_i for y_i , applying α -conversion wherever necessary to avoid captures.

Reduction Semantics of Finite Processes. The above intuition about process behaviour is made precise by the rules in Table [II](#). The *reduction* relation \longrightarrow is the least binary relation on processes satisfying the rules in Table [II](#). The rules are easily seen to realize the above intuition.

We shall use \longrightarrow^* to denote the reflexive, transitive closure of \longrightarrow . A reduction $P \longrightarrow Q$ basically says that P can evolve, after some communication between its subprocesses, into Q . The reductions are quotiented by the *structural congruence* relation \equiv which postulates some basic process equivalences.

Definition 1 (Structural Congruence). Let \equiv be the smallest congruence over processes satisfying the following axioms:

1. $P \equiv Q$ if P and Q differ only by a change of bound names (α -equivalence).
2. $P \mid 0 \equiv P$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$.
3. If $x \notin fn(P)$ then $(\nu x)(P \mid Q) \equiv P \mid (\nu x)Q$.
4. $(\nu x)0 \equiv 0$, $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$.

2.2 Infinite Processes in the Polyadic Pi-Calculus

In the literature there are at least two alternatives to extend the above syntax to express infinite behavior. We describe them next.

Table 1. Reductions Rules

$\text{REACT: } \frac{}{(\cdots + \bar{x}z_1 \cdots z_n.P) \mid (\cdots + x(y_1 \cdots y_n).Q) \longrightarrow P \mid Q\{z_1/y_1, \dots, z_n/y_n\}}$
$\text{PAR: } \frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \qquad \text{RES: } \frac{P \longrightarrow P'}{(\nu x)P \longrightarrow (\nu x)P'}$
$\text{STRUCT: } \frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q}$

Pi with Parametric Recursive Definitions: $\text{p}\pi_{\text{D}}$

A typical way of specifying infinite behavior is by using parametric recursive definitions [Mil99]. In this case we extend the syntax of finite processes (Equation [1](#)) as follows:

$$P, Q, \dots := \dots \mid A(y_1, \dots, y_n) \quad (2)$$

Here $A(y_1, \dots, y_n)$ is an *identifier* (also *call*, or *invocation*) of arity n . We assume that every such an identifier has a unique, possibly recursive, *definition* $A(x_1, \dots, x_n) \stackrel{\text{def}}{=} P$ where the x_i 's are pairwise distinct, and the intuition is that $A(y_1, \dots, y_n)$ behaves as its P with each y_i replacing x_i . We shall presuppose finitely many such definitions. Furthermore, for each $A(x_1, \dots, x_n) \stackrel{\text{def}}{=} P$ we require

$$fn(P) \subseteq \{x_1, \dots, x_n\}. \quad (3)$$

The reduction semantics of the extended processes is obtained simply by extending the structural congruence \equiv in Definition [1](#) with the following axiom:

$$A(y_1, \dots, y_n) \equiv P[y_1, \dots, y_n/x_1, \dots, x_n] \text{ if } A(x_1, \dots, x_n) \stackrel{\text{def}}{=} P. \quad (4)$$

As usual $P[y_1 \dots y_n/x_1 \dots x_n]$ results from syntactically replacing every free occurrence of x_i with y_i and by applying *name α -conversion*, wherever needed to avoid capture.

We shall use $\text{p}\pi_{\text{D}}$ to denote the polyadic π -calculus with parametric recursive definitions with the above syntactic restrictions.

Pi with Replication: $\text{p}\pi_{\text{!}}$

A simple way of expressing infinite behaviour in the π -calculus is by using replication. We shall use $\text{p}\pi_{\text{!}}$ to denote the polyadic π -calculus with replication.

In the $\text{p}\pi_{\text{!}}$ case, the syntax of finite processes (Equation [1](#)) is extended as follows:

$$P, Q, \dots := \dots \mid !P. \quad (5)$$

Intuitively $!P$ behaves as $P \mid P \mid \dots \mid P \mid !P$; unboundedly many copies of P .

The reduction semantics for $\mathfrak{p}\pi_!$ is obtained simply by extending the structural congruence \equiv in Definition 1 with the following axiom:

$$!P \equiv P \mid !P. \tag{6}$$

Barbed Bisimilarity

We shall often state expressiveness results by claiming the existence of a process in one calculus which is equivalent to some given process in another calculus. For this purpose, here we recall a standard way of comparing processes. We shall use $\mathfrak{p}\pi_!$ to denote the calculus with replication.

Let us begin by recalling a basic notion of observation for the π -calculus. Intuitively, given $l = x$ ($l = \bar{x}$) we say that (the barb) l can be *observed* at P , written $P \downarrow_l$, iff P can have an input (output) with subject x . Formally,

Definition 2 (Barbs). Define $P \downarrow_{\bar{x}}$ iff $\exists z, \mathbf{y}, R : P \equiv (\nu z)(\bar{x}\mathbf{y}.Q \mid R)$ and x is not in z . Similarly, $P \downarrow_x$ iff $\exists z, \mathbf{y}, Q, R : P \equiv (\nu z)(x(\mathbf{y}).Q \mid R)$ and x is not in z . Furthermore, $P \Downarrow_l$ iff $\exists Q : P \longrightarrow^* Q \downarrow_l$.

Let us now recall the notion of barbed (weak) bisimilarity and congruence. Remember that a process *context* C in a given calculus is an expression with a hole $[\cdot]$ such that placing a process in the hole produces a well-formed process term in the calculus. If C is a context and P a process, we write $C[P]$ for the process obtained by replacing the $[\cdot]$ in C by P .

For technical purposes, we shall use $\mathfrak{p}\pi_{\mathbb{D}+!}$ as the calculus whose process syntax arises from extending the syntax of finite processes (Equation 1) with both replication and recursive definitions. The reduction semantics of $\mathfrak{p}\pi_{\mathbb{D}+!}$ of the extended processes is obtained by extending the structural congruence \equiv in Definition 1 with the axioms in Equations 4 and 6.

Definition 3 (Barbed Bisimilarity). A (weak) barbed-simulation is a binary relation \mathcal{R} satisfying the following: $(P, Q) \in \mathcal{R}$ implies that:

1. if $P \longrightarrow P'$ then $\exists Q' : Q \longrightarrow^* Q' \wedge (P', Q') \in \mathcal{R}$.
2. if $P \downarrow_l$ then $Q \downarrow_l$.

The relation \mathcal{R} is a barbed bisimulation iff both \mathcal{R} and its converse \mathcal{R}^{-1} are barbed -simulations. We say that P and Q are (weak) barbed bisimilar, written $P \sim Q$, iff $(P, Q) \in \mathcal{R}$ for some barbed bisimulation \mathcal{R} . Furthermore, we say that P and Q are barbed congruent, written $P \approx Q$, iff for each context C in $\mathfrak{p}\pi_{\mathbb{D}+!}$, $C[P] \sim C[Q]$.

2.3 Recursive Definitions vs. Replication in $\mathfrak{P}\mathfrak{i}$

Here we recall a result stating that the variants $\mathfrak{p}\pi_!$ and $\mathfrak{p}\pi_{\mathbb{D}}$ can be regarded as being equally expressive w.r.t (weak) barbed congruence \approx given in Definition 3.

More precisely, the expressiveness criteria w.r.t barbed congruence we shall use in this section can be stated as follows.

Criteria 2. We say that a π -calculus variant is as expressive as another iff for every process P in the second variant one can construct a process $\llbracket P \rrbracket$ in the first variant such that $\llbracket P \rrbracket$ is (weakly) barbed congruent to P .

All the results presented in this section are consequences of the expressiveness results in [SW01].

From $\mathfrak{p}\pi_D$ to $\mathfrak{p}\pi_!$ and Back: Encodings

We shall now provide encodings from one variant into the other and state their correctness. We shall say that a map $\llbracket \cdot \rrbracket$ is a *homomorphism for parallel composition* iff $\llbracket P \mid Q \rrbracket = \llbracket P \rrbracket \mid \llbracket Q \rrbracket$. The notion of homomorphism for the other operators is defined analogously.

Definition 4. Let $\llbracket \cdot \rrbracket_0$ be the map from $\mathfrak{p}\pi_D$ processes and recursive definitions into $\mathfrak{p}\pi_!$ processes given by:

$$\begin{aligned} \llbracket A_i(\mathbf{x}_i) \stackrel{\text{def}}{=} P_i \rrbracket_0 &= ! a_i(\mathbf{x}_i). \llbracket P_i \rrbracket_0, \\ \llbracket A_i(\mathbf{y}_i) \rrbracket_0 &= \bar{a}_i \mathbf{y}_i, \end{aligned}$$

and for all other processes $\llbracket \cdot \rrbracket_0$ is a homomorphism.

Let P be an arbitrary $\mathfrak{p}\pi_D$ process with $\{ A_1(\mathbf{x}_1) \stackrel{\text{def}}{=} P_1, \dots, A_n(\mathbf{x}_n) \stackrel{\text{def}}{=} P_n \}$ as the set of recursive definitions of its process identifiers. The encoding of P , denoted $\llbracket P \rrbracket$, is defined as

$$\llbracket P \rrbracket = (\nu a_1 \cdots a_n)(\llbracket P \rrbracket_0 \mid \prod_{i \in \{1, \dots, n\}} \llbracket A_i(\mathbf{x}_i) \stackrel{\text{def}}{=} P_i \rrbracket_0)$$

where $a_1, \dots, a_n \notin \text{fn}(P)$.

Intuitively, each $A(\mathbf{y})$, with $A(\mathbf{x}) \stackrel{\text{def}}{=} P$, is translated into a particle $\bar{a}\mathbf{y}$ which excites a copy of P (with \mathbf{y} substituted for \mathbf{x}) by interacting with a replicated resource, a provider of instances of P , of the form $! a(\mathbf{x}). \llbracket P \rrbracket$. The correctness of the encoding is stated below.

Theorem 3. Let $\llbracket \cdot \rrbracket$ be the encoding in Definition 4. For each P in $\mathfrak{p}\pi_D$, $P \approx \llbracket P \rrbracket$.

Let us now give an encoding of $\mathfrak{p}\pi_!$ into $\mathfrak{p}\pi_D$. The idea is simple: Each $!P$ is translated into a process A_P , recursively defined as $A_P(\mathbf{x}) \stackrel{\text{def}}{=} P \mid A_P(\mathbf{x})$ which can provide an unbounded number of copies of P .

Definition 5. Let $\llbracket \cdot \rrbracket_0$ be the map from $\mathfrak{p}\pi_!$ processes into $\mathfrak{p}\pi_D$ processes given by:

$$\llbracket !P \rrbracket_0 = A_P(\mathbf{x}) \text{ where } A_P(\mathbf{x}) \stackrel{\text{def}}{=} P \mid A_P(\mathbf{x}) \text{ and } \text{fn}(P) \subseteq \{\mathbf{x}\}$$

and for all other processes $\llbracket \cdot \rrbracket_0$ is a homomorphism.

We can now state the correctness with respect to barbed congruence.

Theorem 4. *Let $[\cdot]$ be the encoding in Definition 5. For each P in $\mathfrak{p}\pi_1$, $P \approx [P]$.*

2.4 Recursion vs. Replication in the Private Pi Calculus

The Private π -calculus [SW01] is a sub-calculus with a restricted form of communication. The idea is that only *bound-outputs* are allowed; i.e, outputs of the form $(\nu z)\bar{x}z.P$. Such bound-outputs are usually abbreviated as $\bar{x}(z)$ assuming that no name occur more than once in z .

The above syntactic restriction results in a pleasant symmetry between inputs and outputs in that they both can be seen as binders. Moreover, the restriction ensures that α -conversion is the only kind of substitution required in the calculus. In fact, the rule REACT in Table 1, which applies a substitution to the continuation of the input, can be replaced by the following rule:

$$\bar{x}(z).P \mid x(z).P \longrightarrow (\nu z)(P \mid Q) \tag{7}$$

Let us denote by $\text{Privp}\pi_1$ the calculus that results from applying to $\mathfrak{p}\pi_1$ the syntactic restriction mentioned above. The $\text{Privp}\pi_D$ calculus is analogously defined as a restriction on $\mathfrak{p}\pi_D$ except that we need an extra-condition to ensure that α -conversion is the only substitution needed in the calculus: In every invocation $A(z)$, no name may occur more than once in the vector z .

Now, if we wish an encoding $[\cdot]$ from $\text{Privp}\pi_1$ into $\text{Privp}\pi_D$ such that $[P] \approx P$, we can simply take that of Definition 5 restricted to the $\text{Privp}\pi_1$ case. As shown below, however, the above restriction makes impossible the existence of an encoding from $\text{Privp}\pi_D$ into $\text{Privp}\pi_1$.

Consider for example the process $P = A(z_0)$ where

$$A(x) \stackrel{\text{def}}{=} \bar{x}(z).A(z).$$

The process P , in parallel with a suitable R , can perform a sequence of actions where the object of an action is the subject of the next one. Sequences of this form are called *logical threads* [SW01]. Moreover, P can perform the infinite logical thread $\bar{z}_0(z_1).\bar{z}_1(z_2).\dots$

Interestingly, as an application of the type theory for $\text{Privp}\pi_1$, the results in [SW01] state that *no process in $\text{Privp}\pi_1$ can exhibit an infinite logical thread*. Together with P above, this property of $\text{Privp}\pi_1$ can be used to prove the following result.

Theorem 5. *There is a process P in $\text{Privp}\pi_D$ such that $P \not\approx Q$ for every Q in $\text{Privp}\pi_1$.*

Therefore, we cannot have an expressiveness result of the kind we have for $\mathfrak{p}\pi_D$ and $\mathfrak{p}\pi_1$ in the previous section. I.e., there is no encoding $[\cdot]$ from $\text{Privp}\pi_D$ processes into $\text{Privp}\pi_1$ processes such that $[P] \approx P$.

3 The Calculus of Communicating Systems (CCS)

Undoubtedly CCS [Mil89], a calculus for synchronous communication, remains as a standard representative of process calculi. In fact, many foundational ideas in the theory of concurrency have sprung from this calculus. In the following we shall consider some variants of CCS without relabelling operations.

3.1 Finite CCS

The finite CCS processes can be obtained as a restriction of the finite processes of the Polyadic π -calculus by requiring all inputs and outputs to have empty subjects only. Intuitively, this means that in CCS there is no sending/receiving of links but synchronization on them. (Notice that the ability of transmitting names is used for the encoding of recursion into replication in Definition 4.) More, precisely, the syntax of finite CCS processes is obtained by replacing the second line of Equation (1) with

$$\alpha := \bar{x} \mid x \mid \tau \tag{8}$$

where τ represents a distinguished action; the *silent* action, with the decree that $\bar{x} = \tau$.

The (unlabelled) reduction relation \longrightarrow for finite CCS processes can be obtained from that for the π -calculus given in the previous section. However, since α -conversion does not hold for one of the CCS variants we consider next, we find it convenient to define \longrightarrow in terms of labelled reduction of CCS given in Table 2. A transition $P \xrightarrow{\alpha} Q$ says that P can perform an action α and evolve into Q . The reduction relation is then defined as $\longrightarrow \stackrel{\text{def}}{=} \tau \rightarrow$.

3.2 Infinite CCS Processes

Both recursion and replication are found in the CCS literature in the forms we saw for the polyadic π -calculus. Nevertheless, as recursion in CCS comes in other forms. Some forms of recursion exhibit *dynamic* name scoping while others, as in the π -calculus, have *static* name scoping. By dynamic scoping we mean that, unlike the static case, the occurrence of a name can get dynamically (i.e., during execution) captured under a restriction. Surprisingly, this will have an impact on their relative expressiveness.

In the literature there are at least four alternatives to extend the above syntax to express infinite behavior. We describe them next.

CCS with Parametric Definitions: CCS_p

The processes of CCS_p calculus are the finite CCS processes plus recursion using parametric definition exactly as in $\text{p}\pi_D$. So in particular we have the restriction

Table 2. An operational semantics for finite CCS

$\text{SUM} \frac{}{\sum_{i \in I} \alpha_i. P_i \xrightarrow{\alpha_j} P_j} \text{ if } j \in I$	$\text{RES} \frac{P \xrightarrow{\alpha} P'}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'} \text{ if } \alpha \notin \{x, \bar{x}\}$	
$\text{PAR}_1 \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$	$\text{PAR}_2 \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$	$\text{COM} \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\bar{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$
$\text{RED} \frac{P \xrightarrow{\tau} Q}{P \longrightarrow Q}$		

on parametric definitions in Equation 3. The calculus is the variant in [Mil99]. The rules for CCS_p are those in Table 2 plus the rule:

$$\text{CALL} \frac{P_A[y_1, \dots, y_n/x_1, \dots, x_n] \xrightarrow{\alpha} P'}{A(y_1, \dots, y_n) \xrightarrow{\alpha} P'} \text{ if } A(x_1, \dots, x_n) \stackrel{\text{def}}{=} P_A \quad (9)$$

As usual $P[y_1 \dots y_n/x_1 \dots x_n]$ results from syntactically replacing every free occurrence of x_i with y_i renaming bound names, i.e., *name α -conversion*, wherever needed to avoid capture. (Of course if $n = 0$, $P[y_1 \dots y_n/x_1 \dots x_n] = P$).

As shown in [Mil99] in CCS_p we can identify process expression differing only by renaming of bound names; i.e., *name α -equivalence*—hence $(\nu x)P$ is the same as $(\nu y)P[y/x]$.

Constant Definitions: CCS_k

We now consider the CCS alternative for infinite behavior given in [Mil89]. We refer to identifiers with arity zero and their corresponding definitions as *constant* and *constant* (or *parameterless*) definitions, respectively. We omit the “()” in $A(\)$.

Given $A \stackrel{\text{def}}{=} P$, requiring all names in $fn(P)$ to be formal parameters, as we did in $\text{p}\pi_D$ (Equation 3), would be too restrictive— P would not have visible actions. Consequently, let us drop the requirement to consider a fragment allowing *only* constant definitions but *with possible occurrence of free names in their bodies*. The rules for this fragments are those of CCS_p . We shall refer to this fragment as CCS_k . In this case Rule CALL, which for CCS_k we prefer to call CONS, takes the form

$$\text{CONS} \frac{P \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'} \text{ if } A \stackrel{\text{def}}{=} P \quad (10)$$

i.e., there is no α -conversion involved; thus allowing name captures. As illustrated in the next section, this causes scoping to be dynamic and α -equivalence not to

hold. This is also the reason we cannot just take the reduction relation \longrightarrow of the π -calculus restricted to CCS_k processes as such a relation assumes α -conversion due to the structural rule.

Recursion Expressions: CCS_μ

Hitherto we have seen process expressions whose recursive behavior is specified in an underlying set of definitions. It is often convenient, however, to have expressions which can specify recursive behavior on their own. Let us now extend the finite CCS processes to include such recursive expressions. The extended syntax is given by:

$$P, Q, \dots := \dots \mid X \mid \mu X.P \tag{11}$$

Here $\mu X.P$ binds the occurrences of the *process variable* X in P . As for bound and free names, the *bound variables* of P , $bv(P)$ are those with a bound occurrence in P , and the *free variables* of P , $fv(P)$ are those with a non-bound occurrence in P . An expression generated by the above syntax is said to be a *process (expression)* iff it is closed (i.e., it contains no free variables). The process $\mu X.P$ behaves as P with the free occurrences of X replaced by $\mu X.P$. Applying *variable* α -conversions wherever necessary to avoid captures. The semantics $\mu X.P$ is given by the rule:

$$\text{REC} \frac{P[\mu X.P/X] \xrightarrow{\alpha} P'}{\mu X.P \xrightarrow{\alpha} P'} \tag{12}$$

We call the resulting calculus CCS_μ . From [EN86] it follows that in CCS_μ we can identify processes up-to name α -equivalence.

Remark 1 (Static and Dynamic Scope: Preservation of α -Equivalence). An interesting issue of the substitution $[\mu X.P/X]$ applied to P is whether it *also requires* the renaming of *bound names* in P to avoid captures (i.e., *name α -conversion*). Such a requirement seems necessary should we want to identify process up-to α -equivalence. In fact, the requirement gives CCS_μ *static* scope of names. Let us illustrate this with an example.

Example 1. Consider $\mu X.P$ with $P = (x \mid (\nu x)(\bar{x}.t \mid X))$. First, let us assume we perform name α -conversions to avoid captures. So, $[\mu X.P/X]$ in P renames the bound x by a fresh name, say z , thus avoiding the capture of P 's free x in the replacement: I.e,

$$P[\mu X.P/X] = (x \mid (\nu z)(\bar{z}.t \mid \mu X.P)) = (x \mid (\nu z)(\bar{z}.t \mid \mu X.(x \mid (\nu x)(\bar{x}.t \mid X))))$$

The reader may care to verify (using the rules in Table 2 plus Rule REC) that t will not be performed; i.e., there is no $\mu X.P \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \dots$ s.t. $\alpha_i = t$.

Now let us assume that the substitution makes no name α -conversion, thus causing a free occurrence of x in P , shown in a box below, to get bound, *dynamically in the scope* of the outermost restriction: I.e.,

$$P[\mu X.P/X] = (x \mid (\nu x)(\bar{x}.t \mid \mu X.P)) = (x \mid (\nu x)(\bar{x}.t \mid \mu X.(\boxed{x} \mid (\nu x)(\bar{x}.t \mid X))))$$

The reader can verify that now t can eventually be performed. Such an execution of t cannot be performed by $\mu X.Q$ where Q is $(x \mid (\nu z)(\bar{z}.t \mid X))$ i.e. P with the binding and bound occurrence of x syntactically replaced with z . This shows that name α -equivalence does not hold in this dynamic scope case. \square

It should be pointed out that using recursive expressions with no name α -conversion is in fact equivalent to using instead constant definitions as in the previous calculus CCS_k . In fact, in presenting CCS, [Mil89] uses alternatively both kinds of constructions; using Rule REC, with no name α -conversion, for one and Rule CONS for the other. For example, by taking $A \stackrel{\text{def}}{=} P$ with P as in Example 1 one can verify that in CCS_k , A exhibits exactly the same dynamic scoping behavior illustrated in the above example. So, *name α -equivalence does not hold in CCS*. Notice that the above observations imply some semantics differences between CCS and the π -calculus. The former does not satisfy name α -equivalence because of the dynamic nature of name scoping—see Example 1. The latter uses static scoping and satisfies α -equivalence. \square

Replication: $CCS_!$

The processes of $CCS_!$ are those finite CCS processes plus replication exactly as in $\pi\pi_!$. This variant is presented in [BGZ03]. In the context of CCS, this operators are studied in [BGZ03, BGZ04, GSV04].

The operational rules for $CCS_!$ are those in Table 2 plus the following rule:

$$\text{REP} \frac{P \mid !P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'} \tag{13}$$

From [Mil99] we know that in $CCS_!$ one can identify processes up to name α -equivalence.

3.3 Expressiveness Results for CCS

In this section we report results from [BGZ03, BGZ04, GSV04] on the expressiveness for the CCS variants above.

The following theorem summarizes the expressiveness of the various calculi and it is an immediate consequence of the results in [BGZ03] and [GSV04]. As for the π -calculus we compare expressiveness w.r.t. barbed congruence with the obvious restriction to CCS contexts (see Criteria 2).

Theorem 6. *The following holds for the CCS variants:*

1. CCS_k is exactly as expressive as CCS_p w.r.t barbed congruence.
2. CCS_μ is exactly as expressive as $CCS_!$ w.r.t barbed congruence.
3. The divergence problem (i.e., whether a given process P has an infinite sequence of \longrightarrow reductions) is undecidable for the calculi in (1) but decidable for those in (2).

The results (1-3) are summarized in Figure 1. Let us now elaborate on the significance and implications of the above results. A noteworthy aspect of (1) is that any finite set of parametric (possibly mutually recursive) definitions can be replaced by a set, *finite* as well, of parameterless definitions. This arises as a result of the restricted nature of communication in CCS (e.g., absence of mobility). Related to this result is that of [Mil89] which shows that, in the context of value-passing CCS, a parametric definition can be encoded using an set of constant definitions and infinite sums. However, this set is *infinite*.

Regarding (1) some readers may feel that given a process P with a parametric definition D , one could simply create as many constant definitions as permutations of possible parameters w.r.t. the finite set of names in P and D . This would not work for CCS_p ; the unfolding of call to D within a restriction may need α -conversions to avoid name captures, thus generating new names (i.e., names not in P nor D) during execution.

Regarding (2), we wish to recall the encoding $\llbracket \cdot \rrbracket$ of CCS_μ into $\text{CCS}_!$ which resembles that of Definition 4 in the context of the π -calculus.

Definition 6. *The encoding $\llbracket \cdot \rrbracket$ of CCS_μ processes into $\text{CCS}_!$ is homomorphic over all operators in the sub-calculus defining finite behavior and is otherwise defined as follows:*

$$\begin{aligned} \llbracket X_i \rrbracket &= \bar{x}_i \\ \llbracket \mu X_i.P \rrbracket &= (\nu x_i)(!x_i.\llbracket P \rrbracket \mid \bar{x}_i) \end{aligned}$$

where the names x_i 's are fresh.

The above encoding is correct w.r.t. barbed congruence, i.e., $\llbracket P \rrbracket \approx P$. It is important to notice that it would not be correct had we adopted dynamic scoping in the Rule REC for CCS_k (see Remark 1). The $\mu X.P$ in Example 1 actually gives us a counter-example.

Another noteworthy aspect of the results mentioned above is the distinction between static and dynamic name scoping for the calculi under consideration. Static scoping renders the calculus with recursion decidable, *w.r.t. the divergence problem*, and no more expressive than the calculus with replication. In contrast, dynamic scoping renders the calculus with constant definitions undecidable and as expressive as that with parametric definitions. This is interesting since as discussed in Section 3.2 the difference between the calculi with static or dynamic scoping is very subtle. Using static scoping for recursive expressions was discussed in the context of ECCS [EN86], an extension of CCS whose ideas lead to the design of the π -calculus [Mil99].

It should be noticed that preservation of divergence is not a requirement for equality of expressiveness w.r.t barbed congruence since *barbed congruence does not preserve divergence*. Hence, although the results in [BGZ03] prove that divergence is decidable for $\text{CCS}_!$ (and undecidable for CCS_p), it does not follow directly from the arrows in Figure 1 that it is also decidable for CCS_μ . The decidability of the divergency problem for CCS_μ is proven in [GSV04].

Finally, it is worth pointing out that, as exposed in [MP03], decidability of divergence does not imply lack of *Turing* expressiveness. In fact a remarkable

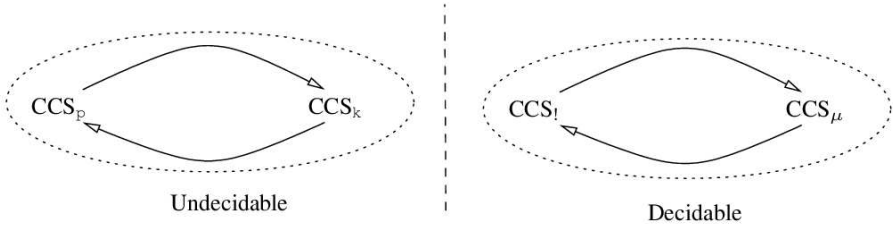


Fig. 1. Classification of CCS variants. An arrow from X to Y indicates that for every P in Y one can construct a process $\llbracket P \rrbracket$ in X which is barbed congruent to P . (Un)decidability is meant w.r.t. the existence of divergent computations.

result in [BGZ04] states that CCS_l is Turing-complete. We shall discuss this at length in the following section.

3.4 CCS_l in the Chomsky Hierarchy

The work in [BGZ04] shows that CCS_l is Turing powerful by encoding Random Access Machines (RAM) in CCS_l . The encoding is said to be *non-deterministic*, or *non-faithful* because it may introduce non-terminating computations which do not correspond to the expected behaviour of the modeled machine. Furthermore, the authors in [BGZ04] also show the non-existence of *deterministic* (or *faithful*) encodings of Turing Machines—i.e., encodings which do not exhibit such additional non-terminating computations.

In [ADNV07] the authors study the expressiveness of CCS_l w.r.t. the existence of faithful encodings of models of computability *strictly less* expressive than Turing Machines. Namely, (non-deterministic) Linear-bounded, Pushdown and Finite-State Automata.

In this section we shall single out the fundamental non-deterministic element for the Turing-expressiveness of CCS_l shown in [BGZ04]. Following [ADNV07] we define a class $\text{CCS}_l^{-\omega}$ of those processes which do not exhibit such kind of non-determinism and discuss their expressiveness w.r.t the Chomsky Hierarchy. First we need a little notation.

Notation 7. Define \xrightarrow{s} , with $s = \alpha_1 \dots \alpha_n \in \mathcal{L}^*$, as

$$(\tau \rightarrow)^* \xrightarrow{\alpha_1} (\tau \rightarrow)^* \dots (\tau \rightarrow)^* \xrightarrow{\alpha_n} (\tau \rightarrow)^*.$$

For the empty sequence $s = \epsilon$, \xrightarrow{s} is defined as $(\tau \rightarrow)^*$.

We say that a process generates a sequence of non-silent actions s if it can perform the actions of s in a finite maximal sequence of transitions. More precisely:

Definition 7 (Sequence and language generation). *The process P generates a sequence $s \in \mathcal{L}^*$ if and only if there exists Q such that $P \xrightarrow{s} Q$ and $Q \not\xrightarrow{\alpha}$ for any $\alpha \in \text{Act}$. Define the language of (or generated by) a process P , $L(P)$, as the set of all sequences P generates.*

Strong non-termination plays a fundamental role in the expressiveness of CCS_!. We borrow the following terminology from rewriting systems:

Definition 8 (Termination). *We say that a process P is (weakly) terminating (or that it can terminate) if and only if there exists a sequence s such that P generates s . We say that P is (strongly) non-terminating, or that it cannot terminate if and only if P cannot generate any sequence.*

As previously mentioned, [BGZ04] provides an encoding $[\![\cdot]\!]$ from Random Access Machines (RAM) into CCS_!. The encoding is called *non-deterministic* in the sense that given a machine M , $[\![M]\!]$ may evolve into states (processes) which do not correspond to any configuration of M . Nevertheless such states are guaranteed to be strongly non-terminating. Therefore, they may be thought of as being configurations which cannot lead to a halting configuration in a non-deterministic Turing machine M' that computes the same function as M .

Now rather than recalling the full encoding of RAMs from [BGZ04], let us use a simpler example which illustrates the same non-deterministic technique. Below we encode a typical context sensitive language in CCS_!.

Example 2. Consider the following processes:

$$\begin{aligned} P &= (\nu k_1, k_2, k_3, u_b, u_c)(\overline{k_1} \mid \overline{k_2} \mid Q_a \mid Q_b \mid Q_c) \\ Q_a &= !k_1.a.(\overline{k_1} \mid \overline{k_3} \mid \overline{u_b} \mid \overline{u_c}) \\ Q_b &= k_1.!k_3.k_2.u_b.b.\overline{k_2} \\ Q_c &= k_2.(!u_c.c \mid u_b.DIV) \end{aligned}$$

where *DIV* is the non-terminating process $(\nu w)(\overline{w} \mid !w.\overline{w})$. It can be verified that $L(P) = \{a^n b^n c^n\}$.

Intuitively, in the process P above, Q_a performs (a sequence of actions) a^n for an arbitrary number n (and also produces n u_b 's). Then Q_b performs b^m for an arbitrary number $m \leq n$ and each time it produces b it consumes a u_b . Finally, Q_c performs c^n and diverges if $m < n$ by checking if there are u_b 's that were not consumed. □

The Power of Non-Termination. Let us underline the role of strong non-termination in Example 2. Consider a run

$$P \xrightarrow{a^n b^m} \dots$$

Observe that the name u_b is used in Q_c to test if $m < n$, by checking whether some u_b were left after generating b^m . If $m < n$, the non-terminating process *DIV* is triggered and the extended run takes the form

$$P \xrightarrow{a^n b^m c^n} \xrightarrow{\tau} \xrightarrow{\tau} \dots$$

Hence the sequence $a^n b^m c^n$ arising from this run (with $m < n$) is therefore not included in $L(P)$.

The tau move. It is crucial to observe that there is a τ transition arising from the moment in which $\overline{k_2}$ chooses to synchronize with Q_c to start performing the c actions. One can verify that if $m < n$ then the process just before that τ transition is weakly terminating while the one just after is strongly non-terminating.

The non-deterministic technique of using non-terminating processes in computing illustrated above is essentially the same given in [BGZ04] for encoding RAM's. Since we want to prevent the use of the above technique, we define a class which avoids moving with a τ transition from a terminating to a non-terminating evolution.

Definition 9 (Weak Termination Preservation and $\text{CCS}_1^{-\omega}$). *A process P is said to be (weakly) termination-preserving (after τ moves) if and only if whenever $P \xrightarrow{s} Q \xrightarrow{\tau} R$: if Q is weakly terminating then R is weakly terminating. Henceforth we use $\text{CCS}_1^{-\omega}$ to denote the set of those CCS_1 processes which are termination-preserving.*

Let us now survey the expressiveness result w.r.t formal language generation (see Definition 7) of the above termination-preserving class of processes $\text{CCS}_1^{-\omega}$.

Language expressiveness of $\text{CCS}_1^{-\omega}$. We assume that the reader is familiar with the notions and notations of formal grammars. A grammar G can be specified as a quad-tuple (Σ, N, S, P) where Σ is the set of terminal symbols, N is the set of non-terminals symbols, S the initial symbol and P the set of production rules. The language of (or generated by) a formal grammar G , denoted as $L(G)$, is defined as all those strings in Σ^* that can be generated by starting with the start symbol S and then applying the production rules in P until no more non-terminal symbols are present. We recall also that in a strictly decreasing expressive order, Types 0, 1, 2 and 3 in the Chomsky hierarchy correspond, respectively, to unrestricted-grammars (Turing Machines), Context Sensitive Grammars (Non-Deterministic Linear Bounded Automata), Context Free Grammars (Non-Deterministic PushDown Automata), and Regular Grammars (Finite State Automata).

The following theorem illustrated in Figure 2 classifies the expressiveness of $\text{CCS}_1^{-\omega}$ in the Chomsky Hierarchy.

Theorem 8. *The following holds for the $\text{CCS}_1^{-\omega}$ variant:*

1. *For every Type 3 Grammar G we can construct a $\text{CCS}_1^{-\omega}$ process P_G such that $L(G) = L(P_G)$.*
2. *There exists a Type 2 Grammar G such that for every $\text{CCS}_1^{-\omega}$ process P_G $L(G) \neq L(P_G)$.*
3. *For every $\text{CCS}_1^{-\omega}$ process P_G there exists a Type 1 Grammar G such that $L(P_G) = L(G)$.*

Let us conclude this section by giving more details on the above classification results. The first point of Theorem 8 follows from a rather straightforward translation from regular expressions.

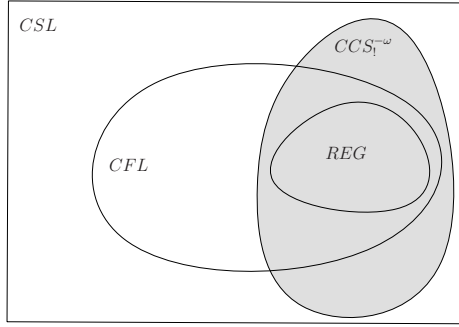


Fig. 2. $CCS_1^{-\omega}$ in the Chomsky Hierarchy

The second one is based on a fundamental property of $CCS_1^{-\omega}$: If $P \in CCS_1^{-\omega}$ can perform an action c after zero or more steps then it can always perform it after a number of steps which depends on the size of P . With the help of this property one can prove that the Type 2 language $a^n b^n c$ cannot be encoded in $CCS_1^{-\omega}$.

The last point follows from a $CCS_1^{-\omega}$ property on the number of τ moves needed to generate any given sequence of size n : It is linearly bounded. More precisely, for every $P \in CCS_1^{-\omega}$, there exists a constant k such that if $s = \alpha_1 \dots \alpha_n \in L(P)$ then there must be a sequence

$$P(\xrightarrow{\tau})^{m_0} \xrightarrow{\alpha_1} (\xrightarrow{\tau})^{m_1} \dots (\xrightarrow{\tau})^{m_{n-1}} \xrightarrow{\alpha_n} (\xrightarrow{\tau})^{m_n} \dashrightarrow$$

with $\sum_{i=0}^n m_i \leq kn$. Using the above property, given P , one can define a non-deterministic machine that simulates the runs of P using as many cells as the total number of performed actions, silent or visible, multiplied by a constant associated to P . Hence $L(P)$ is by definition a Type 1 language.

4 The Mobile Ambients Calculus

The calculus of Mobile Ambients is a formalism for the description of distributed and mobile systems in terms of *ambients*; i.e. a named collection of active processes and nested sub-ambients.

The work in [BZ04] studies the expressiveness of recursion versus replication in Mobile Ambients. In particular, the authors of [BZ04] study the expressive power of ambient mobility in the (Pure) Mobile Ambients variants with replication and recursion.

4.1 Finite Processes of Ambients

The Pure Ambient Calculus focuses on ambient and processes interaction. Unlike the π -calculus, it abstracts away from process communication.

The syntax of the finite processes can be derived from those of the $p\pi$ -calculus by (1) introducing ambients, and the actions for ambient and processes interaction, (2) eliminating the action for process communication and (3) restricting summations to have arity at most one. In summary, we obtain the following syntax:

$$\begin{aligned}
 P, Q, \dots := & 0 \mid \alpha.P \mid n[P] \mid (\nu x)P \mid P \mid Q & (14) \\
 \alpha := & in\ x \mid out\ x \mid open\ x
 \end{aligned}$$

The intuitive behaviour of the ambient $n[P]$ and α actions is better explained after presenting the reduction semantics of Ambients. The intuitive behaviour of the others constructs can be described exactly as in the π -calculus.

Reduction Semantics of Finite Processes. The *reduction* relation \longrightarrow for Ambients can be obtained by adding the axiom $(\nu n)(m[P]) \equiv m[(\nu n)P]$ if $m \neq n$ to the structural congruence in Definition 1 and the following rules for ambients and process interaction to the rules of the $p\pi$ -calculus in Table 1 (without the REACT rule):

1. $n[in\ m.P \mid Q] \mid m[R] \longrightarrow m[n[P \mid Q] \mid R]$
2. $m[out\ m.P \mid Q] \mid R \longrightarrow n[P \mid Q] \mid m[R]$
3. $open\ n.P \mid n[Q] \longrightarrow P \mid Q$
 $\frac{P \longrightarrow Q}{P \longrightarrow Q}$
4. $n[P] \longrightarrow n[Q]$

Rules (1-3) describe ambients and their actions and Rule (4) simply says that reduction can occur underneath ambients. Rule (1) describes how, by using the *in* action, an ambient named n can enter another ambient named m . Similarly, Rule (2) describes how an ambient named n can exit another ambient named m by using the *out* action. Finally Rule (3) describes how a process can dissolve an ambient boundary to access its contents by performing the *open* action over the name n of the ambient.

4.2 Infinite Process of Ambients

Infinite behaviour in Ambients can be represented by using replication as in $p\pi_!$ or recursive expressions of the form $\mu X.P$.

The $MA_!$ Calculus

The calculus $MA_!$ extends the syntax of the finite Ambients processes with $!P$. Its reduction semantics \longrightarrow is obtained by adding the structural axiom $!P \equiv P \mid !P$ to the structural axioms of finite Ambients processes.

The MA_r Calculus

The calculus MA_r extends the syntax of the finite Ambients processes with recursive expression of the form $\mu X.P$ exactly as in CCS_μ (Section 3.2). Its

reduction semantics \longrightarrow is obtained by adding the structural axiom $\mu X.P \equiv P[\mu X.P/X]$ to the structural axioms of finite Ambients processes.

Notice that the issue of the substitution $[\mu X.P/X]$ applied to P we discussed in Section 3.2 arises again: Whether the substitution *also requires* the renaming of *bound names* in P to avoid captures (i.e., *name α -conversion*). Such a requirement seems necessary should we want to identify process up-to *α -equivalence*—which is included in the structural congruence \equiv for Ambients. The CCS examples in Section 3.2 (see Remark 1) can easily be adapted here to illustrate that we obtain dynamic scoping of names if we do not perform the α -conversion in the substitution.

It should be noticed that the above has not been completely clarified in the literature of Ambients. In fact, it raises a technical issue in the results on expressiveness which we shall recall in the next section.

Expressiveness Results

To isolate the expressiveness of restriction and ambient actions in MA_l and MA_r , [BZ04] considers the following fragments of MA_c with $c \in \{l, r\}$: (1) $MA_c^{-\nu}$, the MA_c calculus without the restriction constructor $(\nu x)P$, (2) MA_c^{-mv} , the MA_c calculus without the *in* and *out* actions, and finally (3) $MA_c^{-mv,\nu}$, the corresponding calculus with no *in/out* action nor restriction.

The separation results in [BZ04] among the various calculi are given in terms of the decidability of *termination*; i.e., the problem of whether given a process P does not have any infinite sequence of reductions. Obviously, if the question is decidable in a given calculus then we know that there is no termination-preserving encoding of Turing Machines into the calculus. The results in [BZ04] are summarized in Figure 3.

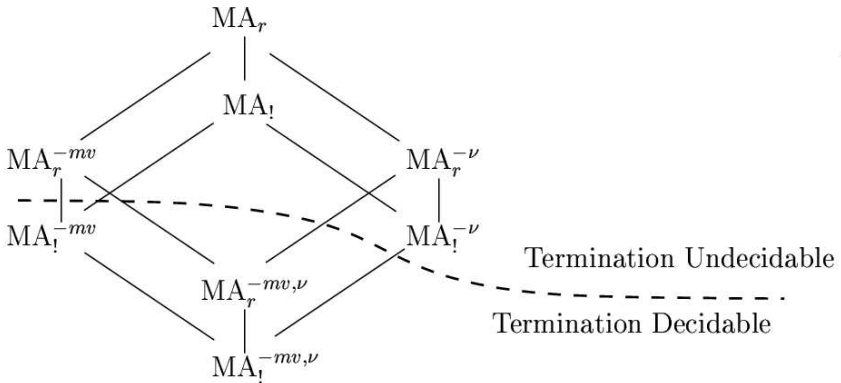


Fig. 3. Hierarchy of Ambient Calculi

Remark 2. The undecidability of process termination for MA_r^{-mv} is obtained by a reduction from termination of RAM machines, a Turing Equivalent formalism.

First [BZ04] uses a CCS fragment with recursion and *dynamic scope of names* to provide a termination-preserving encoding of RAMs. Then the CCS fragment is claimed to be a sub-calculus of MA_r^{-mv} . The undecidability of process termination for MA_r^{-mv} follows immediately.

Nevertheless, as illustrated in Section 3.2 Remark 1 such dynamic scope causes α -equivalence not to be preserved. In principle, this may cause a technical problem in the proof of the result since MA_r^{-mv} requires α -equivalence to be preserved; i.e., the CCS fragment used to simulate RAMs is not a sub-calculus of MA_r^{-mv} .

One way to deal with the above problem is to use a more involved notion of α -conversion in MA_r^{-mv} [BZ05]. Another way would be to consider parametric recursion in MA_r , as in CCS_p or $p\pi_D$, and then use CCS_p as the sub-calculus of MA_r^{-mv} to encode RAMs. Nevertheless, either way we will be changing the original semantics of MA_r^{-mv} given in [LS03] which treats α -conversion and recursion as in CCS_μ [San05].

5 Recursion vs. Replication in Other Calculi

Here, we shall briefly survey work studying the relative expressive power of Recursion vs Replication in other process calculi.

In the context of calculi for security protocols, the work in [HS05] uses a process calculus to analyze the class of ping-pong protocols introduced by Dolev and Yao. The authors show that all nontrivial properties, in particular reachability, become undecidable for a very simple recursive variant of the calculus. The recursive variant is capable of an implicit description of the active intruder, including full analysis and synthesis of messages. The authors then show that the variant with replication renders reachability decidable.

In the context of calculi for Timed Reactive System, the work in [NPV02] studies the expressive power of some variants of Timed concurrent constraint programming (tcc). The tcc model is a process calculus introduced in [SJG94] aimed at specifying timed systems, following the paradigms of Synchronous Languages [BG92]. The work states that: (1) recursive procedures with parameters can be encoded into parameterless recursive procedures with dynamic scoping, and vice-versa. (2) replication can be encoded into parameterless recursive procedures with static scoping, and vice-versa. (3) the languages from (1) are strictly more expressive than the languages from (2). Furthermore, it states that behavioral equivalence is undecidable for the languages from (1), but decidable for the languages from (2). The undecidability result holds even if the process variables take values from a fixed finite domain.

The reader may have noticed the strong resemblance of the work on tcc and that of CCS described in the previous section; e.g., static-dynamic scoping issue w.r.t recursion. In fact, [NPV02] had a great influence in the work we described in this paper for CCS. In particular, in the discovery of the dynamic name scoping exhibited by the CCS presentation in [Mil89].

6 Final Remarks

The expressiveness differences between recursion and replication we have surveyed in this paper may look surprising to those acquainted with the π -calculus where recursion is a derived operation. Our interpretation of this difference is that the link mobility of the π -calculus is a powerful mechanism which makes up for the weakness of replication.

The expressiveness of the replication $!P$ arises from unbounded parallel behaviour, which with recursion can be defined as $\mu X.(P \mid X)$. The additional expressive power of recursion arises from the unbounded nested scope of $\mu X.P$ as in $R = \mu X.(\nu x)(P \mid X)$ which behaves as $(\nu x)(P \mid (\nu x)(P \mid (\nu x)(P \mid \dots)))$. This, in general, cannot be simulated with replication. However, suppose that the unfolding of recursion applies α -conversion to avoid captures as we saw in Section 3.2. For example for the process R above we will have the unfolding $(\nu x_1)(P[x_1/x] \mid (\nu x_2)(P[x_2/x] \mid (\nu x_3)\dots))$ and each x_i will only occur in $P[x_i/x]$. It is easy to see the replication $!(\nu x)P$ captures the behaviour of R . Therefore, R does not really exhibit (significant) unbounded nesting of scope.

All in all, the ability of expressing recursive behaviours via replication in a given process calculus may depend on the mechanisms of the calculus to compensate for the restriction of replication as well as on how meaningful the unbounded nesting of the recursive expressions are.

References

- [ADNV07] Aranda, J., Di Giusto, C., Nielsen, M., Valencia, F.: On the Expressiveness of CCS with Replication. Technical Report, LIX Ecole Polytechnique (2007), URL www.cs.unibo.it/~digiusto/research/ccsrep.pdf
- [BG92] Berry, G., Gonthier, G.: The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming* 19(2), 87–152 (1992)
- [BCMS01] Burkart, O., Caucal, D., Moller, F., Steffen, B.: Verification on infinite structures. In: *Handbook of process algebra*, vol. 9, pp. 545–623. Elsevier, North-Holland (2001)
- [BGZ03] Busi, N., Gabbriellini, M., Zavattaro, G.: Replication vs. recursive definitions in channel based calculi. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) *ICALP 2003*. LNCS, vol. 2719, Springer, Heidelberg (2003)
- [BGZ04] Busi, N., Gabbriellini, M., Zavattaro, G.: Comparing recursion, replication, and iteration in process calculi. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) *ICALP 2004*. LNCS, vol. 3142, Springer, Heidelberg (2004)
- [BZ04] Busi, N., Zavattaro, G.: On the expressive power of movement and restriction in pure mobile ambients. *Theoretical Computer Science* 322(3), 477–515 (2004)
- [BZ05] Busi, N., Zavattaro, G.: Personal Communication (May 2005)
- [CG98] Cardelli, L., Gordon, A.: Mobile Ambients. In: Nivat, M. (ed.) *ETAPS 1998 and FOSSACS 1998*. LNCS, vol. 1378, pp. 140–155. Springer, Heidelberg (1998)

- [EN86] Engberg, U., Nielsen, M.: A calculus of communicating systems with label-passing. Technical report, University of Aarhus (1986)
- [Gir87] Girard, J.-Y.: Linear logic. *Theor. Comput. Sci.* 50, 1–102 (1987)
- [GSV04] Giambiagi, P., Schneider, G., Valencia, F.: On the expressiveness of infinite behavior and name scoping in process calculi. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 226–240. Springer, Heidelberg (2004)
- [Gre78] Greibach, S.A.: Remarks on Blind and Partially Blind One-Way Multi-counter Machines. *Theor. Comput. Sci.* 7, 311–324 (1978)
- [HS05] Huttel, H., Srba, J.: Recursion vs. replication in simple cryptographic protocols. In: Vojtáš, P., Bieliková, M., Charron-Bost, B., Šýkora, O. (eds.) SOFSEM 2005. LNCS, vol. 3381, pp. 175–184. Springer, Heidelberg (2005)
- [LS03] Levi, F., Sangiorgi, D.: Mobile safe ambients. *ACM Transactions on Programming Languages and Systems* 25(1), 1–69 (2003)
- [Mil89] Milner, R.: *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, Englewood Cliffs (1989) (SU Fisher Research 511/24)
- [Mil93] Milner, R.: The polyadic π -calculus: A tutorial. In: Bauer, F.L., Brauer, W., Schwichtenberg, H. (eds.) *Logic and Algebra of Specification*, pp. 203–246. Springer, Berlin (1993)
- [Mil99] Milner, R.: *Communicating and Mobile Systems: the π -calculus*. Cambridge University Press, Cambridge (1999)
- [MP03] Maffeis, S., Phillips, I.: On the computational strength of pure ambient calculi. In: EXPRESS'03 (2003)
- [MPW92] Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, Part I + II. *Information and Computation* 100(1), 1–77 (1992)
- [NPV02] Nielsen, M., Palamidessi, C., Valencia, F.: On the expressive power of concurrent constraint programming languages. In: *Proc. of the 4th International Conference on Principles and Practice of Declarative Programming (PPDP 2002)*, pp. 156–167. ACM Press, New York (2002)
- [Pal97] Palamidessi, C.: Comparing the expressive power of the synchronous and the asynchronous pi-calculus. In: *POPL'97*, pp. 256–265. ACM Press, New York (1997)
- [PV05] Palamidessi, C., Valencia, F.: Recursion vs Replication in Process Calculi. In: *Bulletin of the EATCS* vol. 87, pp. 105–125 (2005)
- [Pet81] Peterson, J.L.: *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Englewood Cliffs (1981)
- [SJJ94] Saraswat, V., Jagadeesan, R., Gupta, V.: Foundations of timed concurrent constraint programming. In: *Proc. of the Ninth Annual IEEE Symposium on Logic in Computer Science, July 4-7, 1994*, pp. 71–80. IEEE Computer Society Press, Los Alamitos (1994)
- [SW01] Sangiorgi, D., Walker, D.: *The π -calculus: A Theory of Mobile Processes*. Cambridge University Press, Cambridge (2001)
- [San05] Sangiorgi, D.: Personal Communication (May 2005)

Bounded Session Types for Object Oriented Languages*

Mariangiola Dezani-Ciancaglini¹, Elena Giachino¹, Sophia Drossopoulou²,
and Nobuko Yoshida²

¹ Dipartimento di Informatica, Università di Torino
`dezani,giachino@di.unito.it`

² Department of Computing, Imperial College London
`scd,yoshida@doc.ic.ac.uk`

Abstract. Earlier work explored the introduction of session types into object oriented languages. Following the session types literature, two parties would start communicating, provided the types attached to that communication, *i.e.* the corresponding session types, were dual of each other. Then, the type system was able to ensure soundness, in the sense that two communicating partners were guaranteed to receive/send sequences of values following the order specified by their session types.

In the current paper we improve upon our earlier work in two ways: we extend the type system to support bounded polymorphism, and we make the selection more object-oriented, so that control structures determine how to continue evaluation, depending on the class of the object being sent/received.

Interestingly, although our notion of selection is more powerful than that in earlier work, the ensuing system turned out not to be more complex, except for the notion of duality, which needed to be extended, to correctly deal with bounded polymorphism, and to capture the new notion of selection.

The paper contains an example, informal explanations, a formal description of the operational semantics and of type system, and a proof of subject reduction.

1 Introduction

In earlier work [12], some of the authors of this paper explored the incorporation of session types [19] into object oriented languages through MOOSE, a minimal object oriented language extended with the notion of a *session*. A session is established when two parties *connect*, using *session types* which are dual to

* This work was partly funded by FP6-2004-510996 Coordination Action TYPES, by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project and by EPSRC GR/T03208, GR/S55538, GR/T04724 and GR/S68071. This paper reflects only the authors' views and the Community is not liable for any use that may be made of the information contained therein.

each other. Session types express sequences and iterations of types of values being received/sent. The *dual* of a session type is a session type where receipt is replaced by sending with a smaller type, and vice versa.

After such a session is established, the two parties can communicate by sending to, or receiving from, each other, objects of the types prescribed in their session type. Through the use of such session types, we could ensure soundness, in the sense that two communicating partners were guaranteed to receive/send values as expected in their session type.

In the current paper we describe the language $\text{MOOSE}_{<}$, which extends MOOSE in the following two ways. Firstly, following ideas from [18,16] we support bounded polymorphism in session types. Thus, the following fragment of a session type

$$?(X<:\text{Image}) . !X$$

expresses that an object of a subclass of `Image` will be received, and then an object of the same class will be sent. This clearly agrees with the standard use of bounded polymorphism for λ -calculus [26].

Secondly, we have made the notion of selection more object-oriented, so that it is possible to determine branch selection and iteration based on the class of the object being sent/received. For example, the following fragment of a session type

$$?(X<:\text{Image}) . ?((Y<:\text{ArrayList}) . !X)^* , (Z<:\text{Image}) . \varepsilon) . ?\text{Address}$$

expresses that first an object of class `X`, a subclass of `Image`, will be received, then a value will be received. If that value is an `ArrayList`, then an `X` will be sent, and this will be repeated until a value of class different from `ArrayList`, *i.e.* `Image`, is received. After that, an `Address` will be received.

A significant part of the design effort for $\text{MOOSE}_{<}$ was devoted to the design of the type system. In order to fully exploit the expressive power of bounded polymorphism we developed a sophisticated notion of duality between session types. The choice based on the class of exchanged objects has required particular care in determining the typing rules.

The type system of MOOSE can also achieve *progress* by taking into account the current channel used to communicate [12]. Because this analysis is orthogonal to the extensions from MOOSE to $\text{MOOSE}_{<}$, and because progress for $\text{MOOSE}_{<}$ could be obtained exactly as for MOOSE , in this work we do not explore this issue any further.

Related Papers. Session types have been proposed first in [19] for the π -calculus and then they have been studied for several different settings, *i.e.* for π -calculus-based formalisms [28,16,20,1,24], for CORBA [29], for functional languages [17,30,11], for object-oriented languages [13,12,10], for boxed ambients [15], and recently, for CDL, a W3C standard description language for web services [32,27,5,21,6]. In this paper we essentially extend the language of [12].

Bounded quantification for object-oriented programming was introduced in [7] as a means of typing functions that operate uniformly over all subtypes of a given type. In order to deal with recursively defined types, bounded quantification was generalised to F-bounded quantification [4]. Pizza [25] is a strict superset of Java that incorporates F-bounded polymorphism. The recently-released version

1.5 of Java adds bounded polymorphism to the language. It is based on a proposal known as GJ (Generic Java) [2]. C# also supports bounded polymorphism. The language PolyTOIL [3] has match-bounded polymorphism that provides a very flexible yet safe type discipline for object-oriented programming. In fact the matching relation is more general than subtyping on object types. We only consider here bounded quantification.

[18] is the first study of bounded polymorphism in the π -calculus, and the first study of any form of polymorphism in relation to session types. In that paper polymorphism is associated with the labels in the branching types. We instead allow to bound all received values in session types.

The selection on the basis of the exchanged object class is reminiscent of the semantic subtyping approach [14,9,8].

Paper structure. We express our ideas through the language MOOSE_<. In Section 2 we give an example, in Sections 3, 4, 5 we give the formal system; in 5.3 we outline soundness, and in the Appendix we give the full proof.

2 Example: Collaborative Card Design

In this section, we describe MOOSE_< through an example, which expresses a typical collaboration pattern, *c.f.* [32,5,6], and which uses our new primitives of bounded polymorphism in session types, and branch selection according to the dynamic type of objects.

This simple protocol contains essential features which demonstrate the expressivity of the new features of MOOSE_<, *i.e.* bounded polymorphism in session types for object oriented languages and branch selection according to the dynamic types of objects.

A card producer and a card customer collaborate for the design of a card that would please the customer. The design is based on two original photos, and some

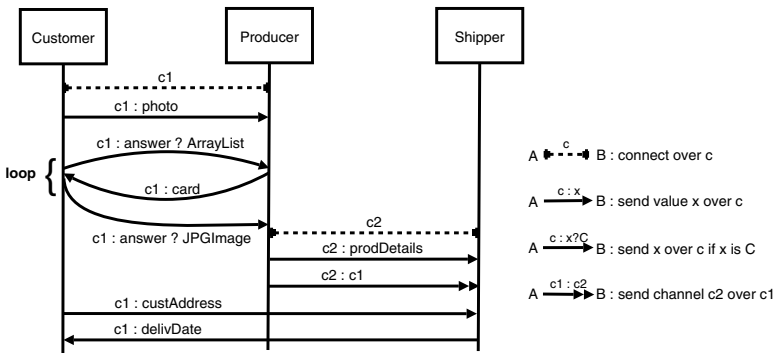


Fig. 1. Collaborative Card Design

card samples sent by the customer. The producer creates a new card based on the customer's originals and samples, and the customer examines the created card, and sends new samples. This process repeats itself (iterates), until the customer is satisfied. The customer expresses his satisfaction (and consequently the end of the iteration) by sending a single image, rather than further card samples. Thus, branch selection in control structures (here iteration) is based on the dynamic type of an object sent.

```

1 session AcceptCards =
2   begin.?(X1<:Image).?(X2<:X1).?(((Y<:ArrayList).!X1)*, (Z<:Image).ε).
3   ?Address.!DeliveryDetails.end
4 session RequestJPGCards =
5   begin.!JPGImage.!JPGImage.!((ArrayList.?(X<:JPGImage))* , Image.ε).
6   !Address.?DeliveryDetails.end
7 session RequestDelivery =
8   begin.!CardSet.!(?Address.!DeliveryDetails.end).end
9 session AcceptDelivery =
10  begin.?CardSet.?(?Address.!DeliveryDetails.end).end

```

Fig. 2. Session Types for the Card Producer-Customer-Shipper Example

Furthermore, the card producer needs to be capable to collaborate with customers who use different image formats. If the customer sends two **JPG** originals, then the card producer should create **JPG** images too. If the customer sends two **GIF** originals, then the card producer should create **GIF** images too. The customer is not allowed to send the two images in different formats, for example one **JPG** and one **GIF** images. We express all this through bounded polymorphism.

In Fig. 1 we show a sequence diagram for the example we described above. The **CardProducer** and **JPGCardCustomer** participants initiate interaction over channel **c1**. The **JPGCardCustomer** sends to the **CardProducer** two photos in **JPG** format followed by his decision, which is either a list of cards (which represent samples and hints for the production of a new card), or just a single, chosen, card. The **CardProducer** examines the decision; if it is a list of cards, then he designs a new card and sends it to the **JPGCardCustomer**, and the process of card design iterates. If the **JPGCardCustomer**'s decision is a single, chosen card, then the iteration terminates, and the **CardProducer** prints the required number of copies of the chosen card. Then the **CardProducer** connects with the **Shipper** over channel **c2** and sends him the printed cards. He then delegates his part of the remaining activity with the **JPGCardCustomer** to the **Shipper**; the latter is realised by sending **c1** over **c2**. Now the **Shipper** will await **JPGCardCustomer**'s address, before responding with the delivery date.

In Fig. 2 we declare the necessary session types, and in Fig. 3, Fig. 4 and Fig. 5 we encode the given scenario in **MOOSE_<**, using one class per protocol participant.

The session types **RequestJPGCards** and **AcceptCards** describe the communication pattern between the **CardProducer** and the **JPGCardCustomer**. The session

type `AcceptCards` describes accepting a first image in *any* format, a second image in the same format of the first one, followed by an iteration. The iteration is repeated as long as the received object is an `ArrayList`, in which case an image, of the same class as the two received is sent; the iteration stops if an `Image` is received¹. After the iteration, an address is received and the delivery details are sent. The session type `RequestJPGCards` models the sending of two images in JPG format, followed by an iteration. The iteration is repeated as long as an `ArrayList` is sent, in which case an `Image` is received; the iteration stops if an `Image` is sent²; afterwards, an address is sent and the delivery details are received. The interesting observation is that (under the assumption that `JPGImage` is a subclass of `Image`) the types `RequestJPGCards` and `AcceptCards` represent *dual* behaviours associated with the same session, in which the sending of a value in one end corresponds to its reception at the other.

Thus, the implementor of `AcceptCards` could also collaborate with a thread which required GIF instead of JPG images. Such a thread would have a type like:

```

1 session RequestGIFCards =
2   begin. !GIFImage. !GIFImage. !((ArrayList.?(X<:GIFImage))* , Image.ε).
3   !Address. ?DeliveryDetails. end

```

In our terminology, `AcceptCards` is a dual of `RequestJPGCards` as well as of `RequestGIFCards`. Thus, in MOOSE_<, more than one session type may be dual of another; therefore, our notion of duality is not standard.

Equally important is the assurance that each value received will belong to the type expected by the receiving party, where the latter can depend on the types of previously exchanged objects. For example, according to `AcceptCards`, the object sent in the communication within the cycle will belong to a subclass of that of the object received in the first communication.

The session type `RequestDelivery` describes sending the printed cards, followed by a *live* session channel of remaining type `?Address. !DeliveryDetails. end`. The session type `AcceptDelivery` is simply `RequestDelivery` with all the external `!` and `?` exchanged.

Sessions can start when two compatible `connect` statements are active. In Fig. 3, the first component of `connect` is the shared channel that is used to start communication, the second is the session type, and the third is the *session body*, which implements the session type. The method `sellCards` of class `CardProducer` contains a `connect` statement that implements the session type `AcceptCards`, while the method `buyCards` of class `JPGCardCustomer` contains a `connect` statement over the same channel and with the session type `RequestJPGCards`. When a `CardProducer` and a `JPGCardCustomer` are executing concurrently the method `sellCards` and `buyCards` respectively, they can engage in a session, which will

¹ The iteration also stops if an object is received which is neither an `ArrayList` nor an `Image`. More in the next sections.

² As before, the iteration also stops if an object is sent which is neither an `ArrayList` nor an `Image`.

```

1  class CardProducer {
2
3      (X <: Image) createCard(X x1, Image x2, ArrayList y) {...} //impl.
        omitted
4
5      void sellCards() {
6          connect c1 AcceptCards {
7              c1.receive(x1) {
8                  c1.receive(x2) {
9                      c1.receiveWhile(y) {
10                         ArrayList ▷ c1.send(createCard(x1, x2, y));
11                         }{ Image ▷ Image card := y;}
12                     } }
13                     CardSet cardSet := cardPrint(card); //impl. omitted
14                     // print the required number of copies of the chosen card
15                     spawn { connect c2 RequestDelivery {
16                         c2.send(cardSet); c2.sendS(c1); } }
17                     } /* End connect */
18                 } /* End method sellCards */
19
20     }

```

Fig. 3. Code for the CardProducer

result in a fresh channel being replaced for occurrences of the shared channel `c1` within both session bodies; freshness guarantees that the new channel only occurs in these two threads, therefore the objects can proceed to perform their interactions without the possibility of external interference.

The type of method `createCard` of the class `CardProducer` is *parameterized*, in that its return type is the class of its first argument, and it is a subclass of the class `Image`. We simplified Java syntax for polymorphic methods in an obvious way, and following the notation from [31].

After starting a session in the body of method `sellCards()`, two photos are received using `c1.receive(x1)`, `c1.receive(x2)` and replace `x1`, `x2`. Then an object is received by the iterative expression `c1.receiveWhile(y)`: while the received object is an array list, a new card – designed out of the photo and of the list of cards – is sent using `c1.send(createCard(x1, x2, y))`. If the received object is an image, the iteration stops. Then, the required copies of this card are printed and a new thread is spawned. The body of the spawn expression has a nested `connect`, via which printed cards are sent to the `Shipper`. Then the actual runtime channel, *i.e.* the channel which substituted `c1` when the outer `connect` took place, is sent through the construct `c2.sendS(c1)`. The latter is an example of *higher-order session communication*.

The method `buyCards` uses the field `cardList` to keep track of the cards proposed by the `CardProducer`. Once the session has started, the two photos are sent using `c1.send(photo1)`, `c1.send(photo2)`, and a card is received using `c1.receive(x)`; this card replaces `x`. The method `examine` takes `cardList` and it

```

1  class JPGCardCustomer {
2
3      JPGImage photo1;
4      JPGImage photo2;
5      ArrayList cardList;
6      Address addr;
7      DeliveryDetails dDetails;
8
9      void buyCards(JPGImage photo) {
10         connect c1 RequestJPGCards {
11             c1.send(photo1);
12             c1.send(photo2);
13             Object answer := examine(cardList); //impl. omitted
14             c1.sendWhile(answer) {
15                 ArrayList ▷ c1.receive(x) {
16                     cardList.add(x);
17                     answer := examine(cardList);}
18                 }{ Image ▷ null; }
19             c1.send(addr);
20             c1.receive(z) { dDetails := z; };
21         } /* End connect */
22     } /* End method buyCards */
23
24 }

```

Fig. 4. Code for the JPGCardCustomer

```

1  class Shipper {
2
3      void delivery() {
4          connect c2 AcceptDelivery {
5              c2.receive(x) { CardSet cardSet := x };
6              c2.receiveS(x) {
7                  x.receive(y) { Address custAddress := y };
8                  DeliveryDetails delivDetails := new DeliveryDetails();
9                  //... set state of delivDetails
10                 x.send(delivDetails);
11             }
12         } /* End connect */
13     } /* End method delivery */
14
15 }

```

Fig. 5. Code for the Shipper

returns an `answer`, which is sent using `c1.sendWhile(answer)`. If the answer is a list of cards (this is meant to happen if the `JPGCardCustomer` does not like any of the proposed cards, and then the answer is meant to contain suggestions

of changes), then a new card is received through `c1.receive(x)` and added to the card list through `cardList.add(x)`, a new `answer` is produced through the call of method `examine`, and the iteration continues. If the answer is an image (this is meant to contain the card chosen by the `JPGCardCostumer`), then the iteration stops. Then, the customer's address, `addr`, is sent and an instance of `DeliveryDetails` is received.

Notice that in order to get an arbitrary number of repetitions, it is crucial to allow objects of different classes to be sent in the different iterations of `sendWhile`.

3 Syntax

In Fig. 6 we describe the syntax of $\text{MOOSE}_{<}$, which extends the language MOOSE [12] to support bounded polymorphism and choice of session communication depending on object classes. We distinguish *user syntax*, *i.e.* source level code, and *runtime syntax*, which includes null pointer exceptions, threads and heaps.

Channels. We distinguish *shared channels* and *live channels*. Shared channels have not yet been connected; they are used to decide if two threads can communicate, in which case they are replaced by fresh live channels. After a connection has been created the channel is live; data may be transmitted through such active channels only.

User syntax. The metavariable t ranges over types for expressions, ρ ranges over running session types, C ranges over class names and s ranges over shared session types. We introduce the full syntax of types in § 5.

Class declarations are as expected, except for the restriction that object fields cannot contain live channels. Without this restriction session would not behave as required, as shown in Example 5.1 of [12].

(type)	$t ::= X \mid C \mid s \mid (s, s)$
(class)	$class ::= class\ C\ extends\ C\ \{ \tilde{f} \tilde{t} \quad \tilde{meth} \}$
(method)	$meth ::= t\ m\ (\tilde{t} \tilde{x}, \tilde{\rho} \tilde{y})\ \{ e \} \mid (X <: t)\ m\ (\tilde{t} \tilde{x}, \tilde{\rho} \tilde{y})\ \{ e \}$
(expression)	$e ::= x \mid v \mid this \mid e; e \mid e.f := e \mid e.f \mid e.m(\tilde{e}) \mid new\ C$ $\quad \mid new\ (s, s) \mid \text{NullExc} \mid spawn\ \{ e \} \mid connect\ u\ s\ \{ e \}$ $\quad \mid u.send(e) \mid u.receive(x)\{e\} \mid u.sendS(u) \mid u.receiveS(x)\{e\}$ $\quad \mid u.sendCase(e)\{\tilde{C} \triangleright \tilde{e}\} \mid u.receiveCase(x)\{\tilde{C} \triangleright \tilde{e}\}$ $\quad \mid u.sendWhile(e)\{\tilde{C} \triangleright \tilde{e}\}\{\tilde{C} \triangleright \tilde{e}\}$ $\quad \mid u.receiveWhile(x)\{\tilde{C} \triangleright \tilde{e}\}\{\tilde{C} \triangleright \tilde{e}\}$
(channel)	$u ::= c \mid x$
(value)	$v ::= c \mid null \mid o$
(thread)	$P ::= e \mid P \mid P$

Fig. 6. Syntax, where syntax occurring only at runtime appears shaded

The method declaration $\mathbf{tm}(\tilde{\mathbf{t}}\tilde{\mathbf{x}}, \tilde{\rho}\tilde{\mathbf{y}})\{e\}$ introduces a standard method, while the method declaration $(X \prec : \mathbf{t})\mathbf{m}(\tilde{\mathbf{t}}\tilde{\mathbf{x}}, \tilde{\rho}\tilde{\mathbf{y}})\{e\}$ introduces a parameterized method whose result has the type of the first of its parameters and this type is bound by \mathbf{t} ³.

The syntax of user expressions e is standard except for the channel constructor $\mathbf{new}(s, s')$, which builds a fresh shared channel used to establish a private session, and the *communication expressions*, i.e. $\mathbf{connect}\ u\ s\{e\}$ and all the expressions in the last three lines.

The first line describes the syntax for parameters, values, the self identifier \mathbf{this} , sequence of expressions, assignment to fields, field access, method call, and object creation. The values are channels and \mathbf{null} . Threads may be created through $\mathbf{spawn}\{e\}$, in which the expression e is called the *thread body*.

The expression $\mathbf{connect}\ u\ s\{e\}$ starts a session: the channel u appears within the term $\{e\}$ in session communications that agree with the session type s . The remaining eight expressions, which realise the exchanges of data, are called *session expressions*, and start with “ $u.$...”; we call u the *subject* of such expressions.

The expressions $u.\mathbf{send}(e)$ and $u.\mathbf{receive}(x)\{e\}$ exchange values (which can be shared channels): the former evaluates e and sends the result over u , while the latter receives a value via u that will be bound to x within e . The expressions $u.\mathbf{sendS}(u')$ and $u.\mathbf{receiveS}(x)\{e\}$ exchange live channels: in $u.\mathbf{receiveS}(x)\{e\}$ the received channel will be bound to x within e , in which x is used for communications.

The next four primitives are extended from those in [12]; they allow choice of communication on the basis of the class of an object sent/received.

The expression $u.\mathbf{sendCase}(e)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}$ first evaluates the expression e to an object o , and sends o over u . It continues with e_i , where i is the smallest index in $\{1, \dots, n\}$ such that o is C_i , if such an i exists. Otherwise, it returns \mathbf{null} . The expression $c.\mathbf{receiveCase}(x)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}$ receives an object o via channel u and binds it to x . It continues with $e_i[o/x]$, where i is the smallest index in $\{1, \dots, n\}$ such that o is C_i , if such an i exists. Otherwise, it returns \mathbf{null} .

The expressions $u.\mathbf{sendWhile}(e)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}\{D_1 \triangleright d_1; \dots; D_m \triangleright d_m\}$ and $u.\mathbf{receiveWhile}(x)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}\{D_1 \triangleright d_1; \dots; D_m \triangleright d_m\}$ express *iterative* communication. The expression $u.\mathbf{sendWhile}(e)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}\{D_1 \triangleright d_1; \dots; D_m \triangleright d_m\}$ evaluates e to an object o and sends it over u . Then it continues with e_i and iterates, where i is the smallest index in $\{1, \dots, n\}$ such that o is C_i , if such i exists. If no such i exists, then it continues with d_j , where j is the smallest index in $\{1, \dots, m\}$ such that o is D_j , if such a j exists. Otherwise, it returns \mathbf{null} . The meaning of the expression $u.\mathbf{receiveWhile}(x)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}\{D_1 \triangleright d_1; \dots; D_m \triangleright d_m\}$ is analogous.

³ We do not expect any technical difficulties in allowing standard parameterized methods. We restricted the result type in this way in order to focus on the features of the object oriented paradigm which interact with sessions.

Runtime syntax. The runtime syntax (shown shaded in Fig. 6) extends the user syntax: it introduces threads running in parallel; adds `NullExc` to expressions, denoting the null pointer error; finally, extends values to allow for object identifiers \mathfrak{o} , which denote references to instances of classes. Single and multiple *threads* are ranged over by P, P' . The expression $P \mid P'$ says that P and P' are running in parallel.

4 Operational Semantics

This section presents the operational semantics of $\text{MOOSE}_{<}$, which is mainly inspired by the language MOOSE of [12]. We only discuss the more interesting rules. First we list the evaluation contexts.

$$E ::= [] \mid E.f \mid E;e \mid E.f := e \mid \mathfrak{o}.f := E \mid E.m(\tilde{e}) \mid \mathfrak{o}.m(\tilde{v}, E, \tilde{e}) \\ \mid c.\text{send}(E) \mid c.\text{sendCase}(E)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}$$

Fig. 7 defines auxiliary functions used in the operational semantics and typing rules. We assume a fixed, global class table CT , which contains *Object* as top-most class.

Objects and fresh channels are stored in *heaps*, whose syntax is given by:

$$h ::= [] \mid h::[\mathfrak{o} \mapsto (C, \tilde{f}:\tilde{v})] \mid h::c$$

Heaps, ranged over h , are built inductively using the heap composition operator “ $::$ ”, and contain mappings of object identifiers to instances of classes, and channels. In particular, a heap will contain the set of objects and *fresh* channels, both shared and live, that have been created since the beginning of execution. The heap produced by composing $h::[\mathfrak{o} \mapsto (C, \tilde{f}:\tilde{v})]$ will map \mathfrak{o} to the object $(C, \tilde{f}:\tilde{v})$, where C is the class name and $\tilde{f}:\tilde{v}$ is a representation for the vector of distinct mappings from field names to their values for this instance. The heap produced by composing $h::c$ will contain the fresh channel c . Heap membership for object identifiers and channels is checked using standard set notation, we therefore write it as $\mathfrak{o} \in h$ and $c \in h$, respectively. Heap update for objects is written $h[\mathfrak{o} \mapsto (C, \tilde{f}:\tilde{v})]$, and field update is written $(C, \tilde{f}:\tilde{v})[f \mapsto v]$.

Expressions. Fig. 8 shows the rules for execution of expressions which correspond to the sequential part of the language. These are identical to the rules of [12] except for the addition of a fresh shared channel to the heap (rule **NewS-R**). In this rule we assume the two session types \mathfrak{s} and \mathfrak{s}' to be *dual*. The duality relation \bowtie will be defined in Fig. 15. In rule **NewC-R** the auxiliary function $\text{fields}(C)$ examines the class table and returns the field declarations for C . The method invocation rule is **Meth-R**; the auxiliary function $\text{mbody}(\mathfrak{m}, C)$ looks up \mathfrak{m} in the class C , and returns a pair consisting of the formal parameter names and the method’s code. The result is the method body where the keyword **this** is replaced by the object identifier \mathfrak{o} , and the formal parameters \tilde{x} are replaced by the actual parameters \tilde{v} .

Field lookup

$$\text{fields}(Object) = \bullet \quad \frac{\text{fields}(D) = \tilde{f}'\tilde{t}' \quad \text{class } C \text{ extends } D \{ \tilde{f} \tilde{t} \tilde{M} \} \in \text{CT}}{\text{fields}(C) = \tilde{f}'\tilde{t}', \tilde{f}\tilde{t}}$$

Method lookup

$$\text{methods}(Object) = \bullet \quad \frac{\text{methods}(D) = \tilde{M}' \quad \text{class } C \text{ extends } D \{ \tilde{f} \tilde{t} \tilde{M} \} \in \text{CT}}{\text{methods}(C) = \tilde{M}', \tilde{M}}$$

Method type lookup

$$\frac{\text{class } C \text{ extends } D \{ \tilde{f} \tilde{t} \tilde{M} \} \in \text{CT} \quad \text{T m } (\tilde{\sigma} \tilde{x}) \{ \mathbf{e} \} \in \tilde{M}}{\text{mtype}(m, C) = \tilde{\sigma} \rightarrow \text{T}}$$

$$\frac{\text{class } C \text{ extends } D \{ \tilde{f} \tilde{t} \tilde{M} \} \in \text{CT} \quad m \notin \tilde{M}}{\text{mtype}(m, C) = \text{mtype}(m, D)}$$

Method body lookup

$$\frac{\text{class } C \text{ extends } D \{ \tilde{f} \tilde{t} \tilde{M} \} \in \text{CT} \quad \text{T m } (\tilde{\sigma} \tilde{x}) \{ \mathbf{e} \} \in \tilde{M}}{\text{mbody}(m, C) = (\tilde{x}, \mathbf{e})}$$

$$\frac{\text{class } C \text{ extends } D \{ \tilde{f} \tilde{t} \tilde{M} \} \in \text{CT} \quad m \notin \tilde{M}}{\text{mbody}(m, C) = \text{mbody}(m, D)}$$

σ is either \mathbf{t} or ρ

T is either \mathbf{t} or $(X <: \mathbf{t})$.

Fig. 7. Lookup Functions

Threads. The reduction rules for threads, shown in Fig. 9, are given modulo the standard structural equivalence rules of the π -calculus [23], written \equiv . We define *multi-step* reduction as: $\longrightarrow^{\text{def}} (\longrightarrow \cup \equiv)^*$. All rules are essentially taken from [12], except for the last three rules which support branching of the communications depending on the class of the object send/received.

When $\text{spawn} \{ \mathbf{e} \}$ is the active redex within an arbitrary evaluation context, the *thread body* \mathbf{e} becomes a new thread, and the original spawn expression is replaced by null in the context.

Rule **Connect-R** describes the opening of sessions: if two threads require a session on the same channel name c with dual session types, then a new fresh channel c' is created and added to the heap. The freshness of c' guarantees privacy and bilinearity of the session communication between the two threads. Finally, the two connect expressions are replaced by their respective session bodies, where the shared channel c has been substituted by the live channel c' .

Rule **ComS-R** gives simple session communication: value v is sent by one thread to another and it is bound to the variable x within the expression \mathbf{e} at the receiver side.

Fld-R $\frac{h(o) = (C, \tilde{f} : \tilde{v})}{o.f_i, h \longrightarrow v_i, h}$	Seq-R $v; e, h \longrightarrow e, h$	FldAss-R $\frac{h' = h[o \mapsto h(o)[f \mapsto v]]}{o.f := v, h \longrightarrow v, h'}$
NewC-R $\frac{\text{fields}(C) = \tilde{f}\tilde{t} \quad o \notin h}{\text{new } C, h \longrightarrow o, h :: [o \mapsto (C, \tilde{f} : \widetilde{\text{null}})]}$	NewS-R $\frac{s \bowtie s' \quad c \notin h}{\text{new } (s, s'), h \longrightarrow c, h :: c}$	
Cong-R $\frac{e, h \longrightarrow e', h'}{E[e], h \longrightarrow E[e'], h'}$	Meth-R $\frac{h(o) = (C, \dots) \quad \text{mbody}(m, C) = (\tilde{x}, e)}{o.m(\tilde{v}), h \longrightarrow e[o/\text{this}][\tilde{y}/\tilde{x}], h}$	
NullProp-R $E[\text{NullExc}], h \longrightarrow \text{NullExc}, h$	NullFldAss-R $\text{null}.f := v, h \longrightarrow \text{NullExc}, h$	
NullFld-R $\text{null}.f, h \longrightarrow \text{NullExc}, h$	NullMeth-R $\text{null}.m(\tilde{v}), h \longrightarrow \text{NullExc}, h$	

Fig. 8. Expression Reduction

Rule **ComSS-R** describes session delegation. One thread is ready to receive a live channel, which will be bound to the variable x within the expression e ; the other thread is ready to send such a channel. Notice that when the channel is exchanged, the receiver spawns a new thread to handle the consumption of the delegated session. This strategy is necessary in order to avoid deadlocks in the presence of circular paths of session delegation, as shown in example 4.2 of [12].

To understand rule **ComSCaseSuccess-R** it is important to notice that in a well-typed process for all $i \in \{1, \dots, n\}$ there is $k \in \{1, \dots, m\}$ such that $C_i <: C'_k$ and vice versa. The sender thread checks if the exchanged object o belongs to some class of its own list of classes: if i is the smallest index in $\{1, \dots, n\}$ such that o is C_i , then the sender thread continues with e_i . Similarly, if k is the smallest index in $\{1, \dots, m\}$ such that o is C'_k , then the receiver thread continues with $e'_k[o/x]$. If instead the object o does not belong to any of the classes C_i, C'_k for $i \in \{1, \dots, n\}$ and $k \in \{1, \dots, m\}$, then rule **ComSCaseFailure-R** is applied: both expressions return `null`. Notice that this choice is made at run time and that it depends on the class of the exchanged object, which it is not known at compile time.

Rule **ComSWhile-R** describes iteration by means of case expressions. The iteration loops on the alternatives in the first pair of lists of classes and expressions, while the second pair of lists represents *default* expressions which can be possibly evaluated only once after the loop end. The typing rules assure that for all $i \in \{1, \dots, n\}$ there is $k \in \{1, \dots, n'\}$ such that $C_i <: C'_k$ and vice versa. Moreover, for all $j \in \{1, \dots, m\}$ there is $l \in \{1, \dots, m'\}$ such that $D_j <: D'_l$ and vice versa. The test is on the class of the exchanged object o : if there exists an i which is the smallest index in $\{1, \dots, n\}$ such that o is C_i , then the sender

Struct

$$P \mid \text{null} \equiv P \quad P \mid P_1 \equiv P_1 \mid P \quad P \mid (P_1 \mid P_2) \equiv (P \mid P_1) \mid P_2 \quad P \equiv P' \Rightarrow P \mid P_1 \equiv P' \mid P_1$$

Spawn-R

$$E[\text{spawn} \{ e \}], h \longrightarrow E[\text{null}] \mid e, h$$

Par-R

$$\frac{P, h \longrightarrow P', h'}{P \mid P_0, h \longrightarrow P' \mid P_0, h'}$$

Str-R

$$\frac{P_1 \equiv P_1 \quad P_1, h \longrightarrow P_2, h' \quad P_2 \equiv P_2'}{P_1, h \longrightarrow P_2', h'}$$

Connect-R

$$\frac{c' \not\prec h \quad s \bowtie s'}{E_1[\text{connect } c \ s \{e_1\}] \mid E_2[\text{connect } c \ s' \{e_2\}], h \longrightarrow E_1[e_1[\mathcal{C}'/c]] \mid E_2[e_2[\mathcal{C}'/c]], h :: c'}$$

ComS-R

$$E_1[u.\text{send}(v)] \mid E_2[u.\text{receive}(x)\{e\}], h \longrightarrow E_1[\text{null}] \mid E_2[e[\forall/x]], h$$

ComSS-R

$$E_1[c.\text{sendS}(c')] \mid E_2[c.\text{receiveS}(x)\{e\}], h \longrightarrow E_1[\text{null}] \mid e[\mathcal{C}'/x] \mid E_2[\text{null}], h$$

ComSCaseSuccess-R

$$\frac{h(o) = (C, \dots) \quad C <: C_i \quad \forall j < i (C \not\prec: C_j) \quad i \in \{1, \dots, n\} \quad C <: C'_k \quad \forall l < k (C \not\prec: C'_l) \quad k \in \{1, \dots, m\}}{E_1[c.\text{sendCase}(o)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}] \mid E_2[c.\text{receiveCase}(x)\{C'_1 \triangleright e'_1; \dots; C'_m \triangleright e'_m\}], h \longrightarrow E_1[e_i] \mid E_2[e'_k[o/x]], h}$$

ComSCaseFailure-R

$$\frac{h(o) = (C, \dots) \quad \forall i \in \{1, \dots, n\} (C \not\prec: C_i) \quad \forall k \in \{1, \dots, m\} (C \not\prec: C'_k)}{E_1[c.\text{sendCase}(o)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}] \mid E_2[c.\text{receiveCase}(x)\{C'_1 \triangleright e'_1; \dots; C'_m \triangleright e'_m\}], h \longrightarrow E_1[\text{null}] \mid E_2[\text{null}], h}$$

ComSWhile-R

$$\frac{E_1[c.\text{sendWhile}(e)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}\{D_1 \triangleright d_1; \dots; D_m \triangleright d_m\}] \mid E_2[c.\text{receiveWhile}(x)\{C'_1 \triangleright e'_1; \dots; C'_{n'} \triangleright e'_{n'}\}\{D'_1 \triangleright d'_1; \dots; D'_{m'} \triangleright d'_{m'}\}], h \longrightarrow E_1[c.\text{sendCase}(e)\{C_1 \triangleright e''_1; \dots; C_n \triangleright e''_n, D_1 \triangleright d_1; \dots; D_m \triangleright d_m\}] \mid E_2[c.\text{receiveCase}(x)\{C'_1 \triangleright e''_1; \dots; C'_{n'} \triangleright e''_{n'}, D'_1 \triangleright d'_1; \dots; D'_{m'} \triangleright d'_{m'}\}], h}{\text{where } e''_i = e_i; c.\text{sendWhile}(e)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}\{D_1 \triangleright d_1; \dots; D_m \triangleright d_m\} \quad e''_{i'} = e_{j'}; c.\text{receiveWhile}(x)\{C'_1 \triangleright e'_1; \dots; C'_{n'} \triangleright e'_{n'}\}\{D'_1 \triangleright d'_1; \dots; D'_{m'} \triangleright d'_{m'}\} \quad (1 \leq i \leq n)(1 \leq j \leq n')}$$

Fig. 9. Thread Reduction

thread continues with e_i , and iterates. Otherwise, if there exists a j which is the smallest index in $\{1, \dots, m\}$ such that o is D_j , then the sender thread continues with d_j . Last, if the object o does not belong to any of the classes C_i, D_j for $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, m\}$, then the sender thread returns null. Dually, if

there exists a k which is the smallest index in $\{1, \dots, n'\}$ such that o is C'_k , then the receiver thread continues with $e'_k[o/x]$, and iterate. Otherwise if there exists an l which is the smallest index in $\{1, \dots, m'\}$ such that o is D'_l , then the receiver thread continues with $d'_l[o/x]$. Last, if the object o does not belong to any of the classes C'_k, D'_l for $k \in \{1, \dots, n'\}$ and $l \in \{1, \dots, m'\}$, then the receiver thread returns null.

5 The Type Assignment System and Its Properties

5.1 Types

The full syntax of types is given in Fig. 10. It extends the syntax of [12] by allowing bounded polymorphism.

Partial session types, ranged over by π , represent sequences of communications, where ε is the empty communication, and $\pi_1.\pi_2$ consists of the communications in π_1 followed by those in π_2 . The partial session types $?(X <: t_1)$ and $!t_2$ express respectively the reception of a value whose type is bound by type t_1 and the sending of a value of type t_2 . Analogously, the partial session types $?(X <: \eta_1)$ and $!(\eta_2)$ represent the exchange of a live channel, and therefore of an active session, with remaining communications bound by η_1 and determined by η_2 , respectively. Note that we allow bounds to be type variables; this supports more expressive session types as shown in the example of Section 2.

The *case types* $!(\tilde{C}.\tilde{\pi})$ and $?((\tilde{X} <: \tilde{C}).\tilde{\pi})$ reflect in an obvious way the structure of the case expressions. The same observation applies to the *iterative types* $!(\tilde{C}.\tilde{\pi})^*$, $\tilde{C}.\tilde{\pi}$ and $?(((\tilde{X} <: \tilde{C}).\tilde{\pi})^*, (\tilde{X} <: \tilde{C}).\tilde{\pi})$ and the while expressions.

An *ended session type*, η , is either a type variable or a partial session type concatenated either with **end** or with a case type whose branches in turn are all ended session types. It expresses a sequence of communications with its termination, *i.e.* no further communications on that channel are allowed at the end. Ended session types guarantee that a channel is consumed, *i.e.* it cannot be further used. This is essential to guarantee the uniqueness of communications in sessions, as shown in Example 5.2 of [12].

$\pi ::= \varepsilon \mid \pi.\pi \mid ?(X <: t) \mid !t \mid ?(X <: \eta) \mid !(\eta) \mid$	
$!(\tilde{C}.\tilde{\pi}) \mid ?((\tilde{X} <: \tilde{C}).\tilde{\pi}) \mid$	
$!(\tilde{C}.\tilde{\pi})^*, \tilde{C}.\tilde{\pi} \mid ?(((\tilde{X} <: \tilde{C}).\tilde{\pi})^*, (\tilde{X} <: \tilde{C}).\tilde{\pi})$	partial session type
$\eta ::= X \mid \pi.\mathbf{end} \mid \pi.\eta \mid !(\tilde{C}.\tilde{\eta}) \mid ?((\tilde{X} <: \tilde{C}).\tilde{\eta})$	ended session type
$\rho ::= \pi \mid \eta$	running session type
$\tau ::= \rho \mid \downarrow$	live session type
$s ::= \mathbf{begin}.\eta$	shared session type
$\theta ::= \tau \mid s$	session type
$t ::= X \mid C \mid s \mid (s, s)$	standard type

Fig. 10. Syntax of Types

Class	Wf-Session	Pair
$C \in \mathcal{D}(\text{CT})$	$\frac{}{\quad}$	$\frac{s \bowtie s'}{\quad}$
$\vdash C : \text{tp}$	$\vdash s : \text{tp}$	$\vdash (s, s') : \text{tp}$

Fig. 11. Well-formed Standard Types

We use ρ to range over both partial session types and ended session types: we call it a *running session type*.

A *live session type* τ is either a running session type or \uparrow . We use \uparrow when typing threads, to indicate the type of a channel which is being used by two threads in complementary ways.

A *shared session type*, s , starts with the keyword **begin** and has one or more endpoints, denoted by **end**. Between the start and each ending point, a sequence of session parts describes the communication protocol. Shared session types are only used to type shared channels which behave as standard values in that they can be sent using **send** and can be stored in object fields. Live channels instead can only be sent using **sendS** and cannot be stored in object fields.

A *session type* θ is either a live session type or a shared session type.

Standard types, t , are either type variables (X), class identifiers (C), shared session types, or pairs of shared session types which are duals (*i.e.* (s, s')). Fig. 11 defines well-formed standard types. Note that $\mathcal{D}(\text{CT})$ denotes the domain of the class table CT , *i.e.* the set of classes declared in CT .

$\Delta := \emptyset \mid \Delta, X <: t \mid \Delta, X <: \rho$		
SuEmp	SuAdd1	SuAdd2
$\frac{}{\emptyset \vdash \text{ok}}$	$\frac{\Delta \vdash \text{ok} \quad X \notin \mathcal{D}(\Delta)}{\Delta, X <: t \vdash \text{ok}}$	$\frac{\Delta \vdash \text{ok} \quad X \notin \mathcal{D}(\Delta)}{\Delta, X <: \rho \vdash \text{ok}}$

Fig. 12. Subtyping Environments

The *subtyping* judgements use subtyping environments which take into account the bounds of type variables. Subtyping environments (ranged over by Δ) are defined in Fig. 12, where $\mathcal{D}(\Delta)$ is the set of the left hand sides in the subtyping judgements of Δ .

The subtyping judgement for standard types has the shape:

$$\Delta \vdash t <: t'$$

and it holds if it can be derived from the axioms of Fig. 13 plus the reflexive and transitive rules. As in [26], we assume that the subclassing is acyclic.

$\frac{}{\Delta, X <: t \vdash X <: t}$	$\frac{\text{class } C \text{ extends } D \{ \tilde{f} \tilde{t} \tilde{M} \} \in \text{CT}}{\Delta \vdash C <: D}$
$\frac{\vdash (s, s') : \text{tp}}{\Delta \vdash (s, s') <: s}$	$\frac{\vdash (s, s') : \text{tp}}{\Delta \vdash (s, s') <: s'}$

Fig. 13. Subtyping for Standard Types

$\frac{\Delta \vdash t <: t'}{\Delta \vdash !t <: !t'}$	$\frac{\Delta \vdash t <: t'}{\Delta \vdash ?(X <: t') <: ?(X <: t)}$
$\frac{\Delta \vdash \eta <: \eta'}{\Delta \vdash !(\eta) <: !(\eta')}$	$\frac{\Delta \vdash \eta <: \eta'}{\Delta \vdash ?(X <: \eta') <: ?(X <: \eta)}$
$\Delta, X <: \eta \vdash X <: \eta$	$\Delta \vdash !(C_1.\rho_1, \dots, C_n.\rho_n) <: !(C_1.\rho'_1, \dots, C_n.\rho'_n)$
$\Delta, X_i <: C_i \vdash \rho_i <: \rho'_i \quad i \in \{1, \dots, n\}$	
$\Delta \vdash ?((X_1 <: C_1).\rho_1, \dots, (X_n <: C_n).\rho_n) <: ?((X_1 <: C_1).\rho'_1, \dots, (X_n <: C_n).\rho'_n)$	
$\Delta \vdash \pi_i <: \pi'_i \quad i \in \{1, \dots, n+m\}$	
$\Delta \vdash !((C_1.\pi_1, \dots, C_n.\pi_n)^*, D_1.\pi_{n+1}, \dots, D_m.\pi_{n+m}) <: !((C_1.\pi'_1, \dots, C_n.\pi'_n)^*, D_1.\pi'_{n+1}, \dots, D_m.\pi'_{n+m})$	
$\Delta, X_i <: C_i \vdash \pi_i <: \pi'_i \quad i \in \{1, \dots, n\} \quad \Delta, Y_j <: D_j \vdash \pi_{n+j} <: \pi'_{n+j} \quad j \in \{1, \dots, m\}$	
$\Delta \vdash ?(((X_1 <: C_1).\pi_1, \dots, (X_n <: C_n).\pi_n)^*, (Y_1 <: D_1).\pi_{n+1}, \dots, (Y_m <: D_m).\pi_{n+m}) <: ?(((X_1 <: C_1).\pi'_1, \dots, (X_n <: C_n).\pi'_n)^*, (Y_1 <: D_1).\pi'_{n+1}, \dots, (Y_m <: D_m).\pi'_{n+m}))$	
$\Delta \vdash \pi <: \pi'$	$\Delta \vdash \pi <: \pi' \quad \Delta \vdash \rho <: \rho'$
$\Delta \vdash \pi.\text{end} <: \pi'.\text{end}$	$\Delta \vdash \pi.\rho <: \pi'.\rho'$

Fig. 14. Subtyping for Running Session Types

The subtyping judgement for running session types has the shape:

$$\Delta \vdash \rho <: \rho'$$

and it holds if it can be derived from the axioms of Fig. 14 plus the reflexive and transitive rules. It is worthwhile to notice that, in contrast with [16], our session subtyping is covariant for outputs and contravariant for inputs with respect to the standard subtyping of the communicated values. The motivation for this is that we expect all channels whose type is a subtype of $!t$ to be able to communicate with channels whose type is $?(X <: t)$ and similarly for the other

$$\begin{array}{c}
\frac{\rho \bowtie \rho'}{\text{begin}.\rho \bowtie \text{begin}.\rho'} \quad \frac{}{\text{end} \bowtie \text{end}} \quad \frac{\rho \bowtie \rho'}{\rho' \bowtie \rho} \quad \frac{\rho \bowtie \rho'[\eta/X]}{!(\eta).\rho \bowtie ?(X <: \eta').\rho'} \quad \frac{\rho \bowtie \rho'[\mathfrak{t}/X]}{\text{!t}.\rho \bowtie ?(X <: \mathfrak{t}').\rho'} \\
\frac{\&_{i \in \{1, \dots, n\}, j \in \{1, \dots, m\}} (\rho_i \bowtie \rho'_j [C_i \curlywedge C'_j / X_j]) \quad \forall i \in \{1, \dots, n\} \exists j \in \{1, \dots, m\}. C_i <: C'_j \quad \forall j \in \{1, \dots, m\} \exists i \in \{1, \dots, n\}. C'_j <: C_i}{!(C_1.\rho_1, \dots, C_n.\rho_n) \bowtie ?((X_1 <: C'_1).\rho'_1, \dots, (X_m <: C'_m).\rho'_m)} \\
\frac{\&_{i \in \{1, \dots, n\}, k \in \{1, \dots, n'\}} (\pi_i \bowtie \pi'_k [C_i \curlywedge C'_k / X_k]) \& \quad \&_{j \in \{1, \dots, m\}, l \in \{1, \dots, m'\}} (\pi_{n+j} \bowtie \pi'_{n'+l} [D_j \curlywedge D'_l / Y_l]) \quad \forall i \in \{1, \dots, n\} \exists k \in \{1, \dots, n'\}. C_i <: C'_k \quad \forall k \in \{1, \dots, n'\} \exists i \in \{1, \dots, n\}. C'_k <: C_i \quad \forall j \in \{1, \dots, m\} \exists l \in \{1, \dots, m'\}. D_j <: D'_l \quad \forall l \in \{1, \dots, m'\} \exists j \in \{1, \dots, m\}. D'_l <: D_j}{!(C_1.\pi_1, \dots, C_n.\pi_n)^*, D_1.\pi_{n+1}, \dots, D_m.\pi_{n+m}) \bowtie \quad ?(((X_1 <: C'_1).\pi'_1, \dots, (X_{n'} <: C'_{n'}).\pi'_{n'})^* (Y_1 <: D'_1).\pi'_{n'+1}, \dots, (Y_{m'} <: D'_{m'}).\pi'_{n'+m'})} \\
\end{array}$$

Fig. 15. Duality Relation

types. This requires the given rules for output. Similar reasons justify the rules for inputs.

In order to guarantee protocol soundness it is crucial to introduce a duality relation between shared session types which ensures that two sessions agree with each other with respect to the order in which data are communicated and with respect to the types of the communicated data. The introduction of bounded polymorphism allows to relate more than one session type to another session type, in contrast to the systems of [1,20,30,16,12], where each session type has a unique dual. The definition of this relation (denoted by \bowtie) is given in Fig. 15, where we use $\mathfrak{t} <: \mathfrak{t}'$ as shorthand for $\emptyset \vdash \mathfrak{t} <: \mathfrak{t}'$ and similarly for $\rho <: \rho'$.

Two case types $!(C_1.\rho_1, \dots, C_n.\rho_n)$ and $?((X_1 <: C'_1).\rho'_1, \dots, (X_m <: C'_m).\rho'_m)$ are dual if for any arbitrary class C either $C <: C_i$ and $C <: C'_j$ for some $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, m\}$, or $C \not<: C_i$ and $C \not<: C'_j$ for all $i \in \{1, \dots, n\}$ and $j \in \{1, \dots, m\}$. This condition is necessary in order to ensure applicability of one of the two reduction rules **ComSCaseSuccess-R** and **ComSCaseFailure-R**. We ensure this condition by requiring that for all $i \in \{1, \dots, n\}$ there exists a $j \in \{1, \dots, m\}$ such that $C_i <: C'_j$ and for all $j \in \{1, \dots, m\}$ there exists a $i \in \{1, \dots, n\}$ such that $C'_j <: C_i$. Moreover, if there is a class which is a subclass of both C_i and C_j (i.e. C_i and C_j are comparable), then we need to guarantee that the two threads will continue the communication in a coherent way⁴. This means that ρ_i and ρ'_j must agree as specified below. Instead, if C_i and C_j are incomparable, then ρ_i and ρ'_j can be unrelated. In order to express the above conditions we define the “minimum” of two classes as follows:

⁴ Taking into account the order in which the classes appear we could avoid to check some pairs of i, j . For simplicity we do not consider this refinement.

$$C \curlywedge C' = \begin{cases} C & \text{if } C <: C', \\ C' & \text{if } C' <: C, \\ \perp & \text{otherwise.} \end{cases}$$

and we require $\rho_i \bowtie \rho'_j[C_i \curlywedge C'_j/X_j]$ for all $i \in \{1, \dots, n\}$ and all $j \in \{1, \dots, m\}$, with the convention $\rho_i \bowtie \rho'_j[\perp/X_j] = \text{true}$.

The duality of iterative types can be explained similarly, taking into account that we need to ensure that either both threads choose to iterate, or both threads choose default expressions, or both threads choose null.

It is easy to verify by induction on the definition of \bowtie that subtyping preserves duality, *i.e.*:

Lemma 5.1. *If $\rho \bowtie \rho'$ and $\rho'' <: \rho'$, then $\rho \bowtie \rho''$.*

5.2 Typing Rules

The typing judgements for expressions and threads have three environments, *i.e.* they have the shape:

$$\Delta; \Gamma; \Sigma \vdash e : t \qquad \Delta; \Gamma; \Sigma \vdash P : \text{thread}$$

where the *standard environment* Γ associates standard types to this, parameters, objects, and shared channels, while the *session environment* Σ contains

Standard Environments, and Well-formed Standard Environments

$$\Gamma ::= \emptyset \mid \Gamma, x : t \mid \Gamma, \text{this} : C \mid \Gamma, o : C \mid \Gamma, c : s \mid \Gamma, c : (s, s')$$

Emp

$$\frac{\emptyset \vdash \text{ok}}{\text{EOid}} \quad \frac{\Gamma \vdash \text{ok} \quad C \in \mathcal{D}(\text{CT}) \quad o \notin \mathcal{D}(\Gamma)}{\Gamma, o : C \vdash \text{ok}}$$

EChal

$$\frac{\Gamma \vdash \text{ok} \quad \vdash s : \text{tp} \quad c \notin \mathcal{D}(\Gamma)}{\Gamma, c : s \vdash \text{ok}}$$

EVar

$$\frac{\Gamma \vdash \text{ok} \quad \vdash t : \text{tp} \quad x \notin \mathcal{D}(\Gamma)}{\Gamma, x : t \vdash \text{ok}}$$

Ethis

$$\frac{\Gamma \vdash \text{ok} \quad C \in \mathcal{D}(\text{CT}) \quad \text{this} \notin \mathcal{D}(\Gamma)}{\Gamma, \text{this} : C \vdash \text{ok}}$$

ECha2

$$\frac{\Gamma \vdash \text{ok} \quad \vdash (s, s') : \text{tp} \quad c \notin \mathcal{D}(\Gamma)}{\Gamma, c : (s, s') \vdash \text{ok}}$$

Session Environments, and Well-formed Session Environments

$$\Sigma ::= \emptyset \mid \Sigma, u : \theta$$

SEmp

$$\frac{}{\emptyset \vdash \text{ok}}$$

SeAdd

$$\frac{\Sigma \vdash \text{ok} \quad u \notin \mathcal{D}(\Sigma)}{\Sigma, u : \theta \vdash \text{ok}}$$

Fig. 16. Standard and Session Environments

only judgements for channel names and variables. Fig. 16 defines well-formedness of standard and session environments, where the domain of an environment is defined as usual and denoted by $\mathcal{D}()$.

The main differences with the typing rules of [12] are the addition of bounded polymorphism and the deletion of hot sets, which in [12] were used to guarantee progress, a property we will not consider here. As we already discussed in the introduction, we could add hot sets to get progress without problems.

Typing Rules for Values

$$\begin{array}{c}
 \text{Null} \\
 \frac{\Gamma \vdash \text{ok} \quad \vdash t : \text{tp}}{\Delta; \Gamma; \emptyset \vdash \text{null} : t} \\
 \\
 \text{Oid} \\
 \frac{\Gamma, o : C \vdash \text{ok}}{\Delta; \Gamma, o : C; \emptyset \vdash o : C} \\
 \\
 \text{Chan} \\
 \frac{\Gamma, c : t \vdash \text{ok}}{\Delta; \Gamma, c : t; \emptyset \vdash c : t}
 \end{array}$$

Typing Rules for Standard Expressions

$$\begin{array}{c}
 \text{Var} \\
 \frac{\Gamma, x : t \vdash \text{ok}}{\Delta; \Gamma, x : t; \emptyset \vdash x : t} \\
 \\
 \text{This} \\
 \frac{\Gamma, \text{this} : C \vdash \text{ok}}{\Delta; \Gamma, \text{this} : C; \emptyset \vdash \text{this} : C} \\
 \\
 \text{Fld} \\
 \frac{\Delta; \Gamma; \Sigma \vdash e : C \quad \text{ft} \in \text{fields}(C)}{\Delta; \Gamma; \Sigma \vdash e.f : t} \\
 \\
 \text{Seq} \\
 \frac{\Delta; \Gamma; \Sigma \vdash e : t \quad \Delta; \Gamma; \Sigma' \vdash e' : t'}{\Delta; \Gamma; \Sigma \circ \Sigma' \vdash e; e' : t'} \\
 \\
 \text{FldAss} \\
 \frac{\Delta; \Gamma; \Sigma \vdash e : C \quad \Delta; \Gamma; \Sigma' \vdash e' : t \quad \text{ft} \in \text{fields}(C)}{\Delta; \Gamma; \Sigma \circ \Sigma' \vdash e.f := e' : t} \\
 \\
 \text{NewC} \\
 \frac{\Gamma \vdash \text{ok} \quad C \in \mathcal{D}(\text{CT})}{\Delta; \Gamma; \emptyset \vdash \text{new } C : C} \\
 \\
 \text{NewS} \\
 \frac{\Gamma \vdash \text{ok}}{\Delta; \Gamma; \emptyset \vdash \text{new } (s, s') : (s, s')} \\
 \\
 \text{Spawn} \\
 \frac{\Delta; \Gamma; \Sigma \vdash e : t \quad \text{ended}(\Sigma)}{\Delta; \Gamma; \Sigma \vdash \text{spawn } \{ e \} : \text{Object}} \\
 \\
 \text{NullPE} \\
 \frac{\Gamma \vdash \text{ok} \quad \vdash t : \text{tp}}{\Delta; \Gamma; \emptyset \vdash \text{NullExc} : t} \\
 \\
 \text{Meth} \\
 \frac{\Delta; \Gamma; \Sigma_0 \vdash e : C \quad \Delta; \Gamma; \Sigma_i \vdash e_i : t_i \quad i \in \{1 \dots n\} \\
 \text{mtype}(m, C) = t_1, \dots, t_n, \rho_1, \dots, \rho_m \rightarrow t}{\Delta; \Gamma; \Sigma_0 \circ \Sigma_1 \dots \circ \Sigma_n \circ \{u_1 : \rho_1, \dots, u_m : \rho_m\} \vdash e.m(e_1, \dots, e_n, u_1, \dots, u_m) : t} \\
 \\
 \text{MethB} \\
 \frac{\Delta; \Gamma; \Sigma_0 \vdash e : C \quad \Delta; \Gamma; \Sigma_i \vdash e_i : t_i \quad i \in \{1 \dots n\} \\
 \text{mtype}(m, C) = X, t_2, \dots, t_n, \rho_1, \dots, \rho_m \rightarrow X <: t \quad \Delta \vdash t_1 <: t}{\Delta; \Gamma; \Sigma_0 \circ \Sigma_1 \dots \circ \Sigma_n \circ \{u_1 : \rho_1, \dots, u_m : \rho_m\} \vdash e.m(e_1, \dots, e_n, u_1, \dots, u_m) : t_1}
 \end{array}$$

Fig. 17. Typing Rules for Expressions I

Typing Rules for Communication Expressions

$$\frac{\text{Conn} \quad \Delta; \Gamma; \emptyset \vdash u : \text{begin}.\eta \quad \Delta; \Gamma \setminus u; \Sigma, u : \eta \vdash e : t}{\Delta; \Gamma; \Sigma \vdash \text{connect } u \text{ begin}.\eta \{e\} : t}$$

$$\frac{\text{Send} \quad \Delta; \Gamma; \Sigma \vdash e : t}{\Delta; \Gamma; \Sigma \circ \{u : !t\} \vdash u.\text{send}(e) : \text{Object}}$$

$$\frac{\text{Receive} \quad \Delta, X <: t; \Gamma, x : X; \Sigma \vdash e : t' \quad X \notin \Gamma \cup \Delta \cup \Sigma \setminus u}{\Delta; \Gamma; \{u : ?(X <: t)\} \circ \Sigma \vdash u.\text{receive}(x)\{e\} : t'}$$

$$\frac{\text{SendS} \quad \Gamma \vdash \text{ok} \quad \eta \neq \varepsilon.\text{end}}{\Delta; \Gamma; \{u' : \eta, u : !(\eta)\} \vdash u.\text{sendS}(u') : \text{Object}}$$

$$\frac{\text{ReceiveS} \quad \Delta, X <: \eta; \Gamma \setminus x; \{x : X\} \vdash e : t \quad \eta \neq \varepsilon.\text{end}}{\Delta; \Gamma; \{u : ?(X <: \eta)\} \vdash u.\text{receiveS}(x)\{e\} : \text{Object}}$$

$$\frac{\text{SendCase} \quad \Delta; \Gamma; \Sigma_0 \vdash e : C \quad \Delta; \Gamma; \Sigma, u : \rho_i \vdash e_i : t \quad C_i <: C \quad i \in \{1, \dots, n\}}{\Delta; \Gamma; \Sigma_0 \circ \Sigma, u : !(\langle C_1.\rho_1, \dots, C_n.\rho_n \rangle) \vdash u.\text{sendCase}(e)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\} : t}$$

$$\frac{\text{ReceiveCase} \quad \Delta, X_i <: C_i; \Gamma, x : X_i; \Sigma, u : \rho_i \vdash e_i : t \quad i \in \{1, \dots, n\}}{\Delta; \Gamma; \Sigma, u : ?(\langle X_1 <: C_1, \rho_1, \dots, X_n <: C_n, \rho_n \rangle) \vdash u.\text{receiveCase}(x)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\} : t}$$

$$\frac{\text{SendWhile} \quad \Delta; \Gamma; \emptyset \vdash e : C \quad \Delta; \Gamma; \{u : \pi_i\} \vdash e_i : t \quad C_i <: C \quad i \in \{1, \dots, n\}}{\Delta; \Gamma; \{u : \pi_{n+j}\} \vdash d_j : t \quad D_j <: C \quad j \in \{1, \dots, m\}}}{\Delta; \Gamma; \{u : !(\langle C_1.\pi_1, \dots, C_n.\pi_n \rangle^*, D_1.\pi_{n+1}, \dots, D_m.\pi_{n+m})\}} \vdash u.\text{sendWhile}(e)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}\{D_1 \triangleright d_1; \dots; D_m \triangleright d_m\} : t$$

$$\frac{\text{ReceiveWhile} \quad \Delta, X_i <: C_i; \Gamma, x : X_i; \{u : \pi_i\} \vdash e_i : t \quad i \in \{1, \dots, n\}}{\Delta, Y_j <: D_j; \Gamma, x : Y_j; \{u : \pi_{n+j}\} \vdash d_j : t \quad j \in \{1, \dots, m\}}}{\Delta; \Gamma; \{u : ?(\langle X_1 <: C_1, \pi_1, \dots, X_n <: C_n, \pi_n \rangle^*, Y_1 <: D_1, \pi_{n+1}, \dots, Y_m <: D_m, \pi_{n+m})\}} \vdash u.\text{receiveWhile}(x)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}\{D_1 \triangleright d_1; \dots; D_m \triangleright d_m\} : t$$

Non-structural Typing Rules for Expressions

$$\frac{\text{Sub} \quad \Delta; \Gamma; \Sigma \vdash e : t \quad \Delta \vdash t <: t'}{\Delta; \Gamma; \Sigma \vdash e : t'} \quad \frac{\text{WeakES} \quad \Delta; \Gamma; \Sigma \vdash e : t \quad u \notin \mathcal{D}(\Sigma)}{\Delta; \Gamma; \Sigma, u : \varepsilon \vdash e : t} \quad \frac{\text{WeakE} \quad \Delta; \Gamma; \Sigma, u : \pi \vdash e : t}{\Delta; \Gamma; \Sigma, u : \pi.\text{end} \vdash e : t}$$

Fig. 18. Typing Rules for Expressions II

$\frac{\text{M-ok} \quad \emptyset; \{\text{this}: C, \tilde{x}: \tilde{t}\}; \{\tilde{y}: \tilde{\rho}\} \vdash e: t}{t \text{ m } (\tilde{t} \tilde{x}, \tilde{\rho} \tilde{y}) \quad \{e\}: \text{ok in } C}$	$\frac{\text{MS-ok} \quad \{X <: t\}; \{\text{this}: C, x: X, \tilde{x}: \tilde{t}\}; \{\tilde{y}: \tilde{\rho}\} \vdash e: X}{(X <: t) \text{ m } (Xx, \tilde{t} \tilde{x}, \tilde{\rho} \tilde{y}) \quad \{e\}: \text{ok in } C}$
$\frac{\text{C-ok} \quad \tilde{M}: \text{ok in } C}{\text{class } C \text{ extends } D \quad \{\tilde{f} \tilde{t} \tilde{M}\}: \text{ok}}$	$\frac{\text{CT-ok} \quad \text{class } C \text{ extends } D \quad \{\tilde{f} \tilde{t} \tilde{M}\}: \text{ok} \quad \text{CT}: \text{ok}}{\text{CT}, \text{class } C \text{ extends } D \quad \{\tilde{f} \tilde{t} \tilde{M}\}: \text{ok}}$

Fig. 19. Well-formed Class Tables

$\frac{\text{Start} \quad \Delta; \Gamma; \Sigma \vdash e: t}{\Delta; \Gamma; \Sigma \vdash e: \text{thread}}$	$\frac{\text{Par} \quad \Delta; \Gamma; \Sigma_i \vdash P_i: \text{thread} \quad i \in \{1, 2\}}{\Delta; \Gamma; \Sigma_1 \parallel \Sigma_2 \vdash P_1 \mid P_2: \text{thread}}$
--	---

Fig. 20. Typing Rules for Threads

In Fig. 17, Fig. 18 and Fig. 20 we give the typing rules for expressions and threads using the lookup functions defined in Fig. 7. In the typing rules for expressions the session environments of the conclusions are obtained from those of the premises and possibly other session environments using the *concatenation* operator, \circ , defined below. We consider different cases for the concatenation of session types since we want to avoid to have redundant ε . As usual, \perp stands for undefined.

$$- \rho \circ \rho' = \begin{cases} \rho & \text{if } \rho' = \varepsilon \\ \rho' & \text{if } \rho = \varepsilon \\ \rho.\text{end} & \text{if } \rho' = \varepsilon.\text{end} \text{ and } \rho \text{ is a partial session type} \\ \rho.\rho' & \text{if } \rho \text{ is a partial session type and} \\ & \rho' \text{ is a running session type} \\ \perp & \text{otherwise.} \end{cases}$$

$$- \Sigma \setminus \Sigma' = \{u: \Sigma(u) \mid u \in \mathcal{D}(\Sigma) \setminus \mathcal{D}(\Sigma')\}$$

$$- \Sigma \circ \Sigma' = \begin{cases} \Sigma \setminus \Sigma' \cup \Sigma' \setminus \Sigma \cup \{u: \Sigma(u) \circ \Sigma'(u) \mid u \in \mathcal{D}(\Sigma) \cap \mathcal{D}(\Sigma')\} & \text{if } \forall u \in \mathcal{D}(\Sigma) \cap \mathcal{D}(\Sigma'): \Sigma(u) \circ \Sigma'(u) \neq \perp; \\ \perp & \text{otherwise.} \end{cases}$$

The concatenation of two channel types ρ and ρ' is the unique channel type (if it exists) which prescribes all the communications of ρ followed by all those of ρ' . The concatenation only exists if ρ is a partial session type, and ρ' is a running session type. The extension to session environments is straightforward. The typing rules concatenate the session environments to take into account the

order of execution of expressions. We adopt the convention that typing rules are applicable only when the session environments in the conclusions are defined.

Rule **Spawn** requires that all sessions used by the spawned thread are finally consumed, *i.e.* they are all ended session types. This is necessary in order to avoid configurations in which more than two threads are ready to communicate on the same live channel. To guarantee the consumption we define:

$$\text{ended}(\Sigma) = \forall \mathbf{u} : \rho \in \Sigma. \rho \text{ is an ended session type.}$$

Rules **Meth** and **MethB** retrieve the type of the method \mathbf{m} from the class table using the auxiliary function $\text{mtype}(\mathbf{m}, C)$ defined in Fig. 7. The session environments of the premises are concatenated with $\{\mathbf{u}_1 : \rho_1, \dots, \mathbf{u}_m : \rho_m\}$, which represents the communication protocols of the channels $\mathbf{u}_1, \dots, \mathbf{u}_m$ during the execution of the method body. The difference between the two rules is the type of the return value, which is fixed in rule **Meth** and instead parametrized on the type of the first argument in rule **MethB**.

Rule **Conn** ensures that a session body properly uses its unique channel according to the required session type. The first premise says that the shared channel used for the session (\mathbf{u}) can be typed with the appropriate shared session type ($\text{begin}.\eta$). The second premise ensures that the session body can be typed in the restricted standard environment $\Gamma \setminus \mathbf{u}$ with a session environment containing $\mathbf{u} : \eta$.

Rules **SendCase** and **ReceiveCase** put together the types of the different alternatives in the expected way. Notice that, in a specific case expression, all ρ_i for $i \in \{1, \dots, n\}$ are either partial session types or ended session types – this is guaranteed by the syntax of case session types. Similarly for rules **SendWhile** and **ReceiveWhile**.

Rule **WeakES**, where **ES** stands for empty session, is necessary to type a branch of a case expression where the channel which is the subject of the conditional is not used. Rule **WeakE**, where **E** stands for end, allows us to obtain ended session types as predicates of session environments in order to apply rules **Conn**, **Spawn** and **ReceiveS**.

Fig. 19 defines well-formed class tables. Rules **M-ok** and **MS-ok** type-check the method bodies with respect to a class C taking as environments the association between formal parameters and their types and the association between this and C . These rules differ in the return type of the method.

In the typing rules for threads, we need to take into account that the same channel can occur with dual types in the session environments of two premises. For this reason we compose the session environments of the premises using the *parallel composition*, \parallel . We define parallel composition, \parallel , on session types and on session environments as follows:

$$\theta \parallel \theta' = \begin{cases} \uparrow & \text{if } \theta \bowtie \theta' \\ \perp & \text{otherwise.} \end{cases}$$

$$\Sigma \parallel \Sigma' = \begin{cases} \Sigma \setminus \Sigma' \cup \Sigma' \setminus \Sigma \cup \{\mathbf{u} : \Sigma(\mathbf{u}) \parallel \Sigma'(\mathbf{u}) \mid \mathbf{u} \in \mathcal{D}(\Sigma) \cap \mathcal{D}(\Sigma')\} & \text{if } \forall \mathbf{u} \in \mathcal{D}(\Sigma) \cap \mathcal{D}(\Sigma') : \Sigma(\mathbf{u}) \parallel \Sigma'(\mathbf{u}) \neq \perp \\ \perp & \text{otherwise.} \end{cases}$$

$\frac{\mathbf{HNull} \quad C \in \mathcal{D}(\mathbf{CT})}{\Gamma; h \vdash \text{null} : C}$	$\frac{\mathbf{HObj} \quad h(\mathfrak{o}) = (\Gamma(\mathfrak{o}), \dots) \quad \emptyset \vdash \Gamma(\mathfrak{o}) <: C}{\Gamma; h \vdash \mathfrak{o} : C}$	$\frac{\mathbf{HCha} \quad \emptyset \vdash \Gamma(\mathfrak{c}) <: \mathfrak{t}}{\Gamma; h \vdash \mathfrak{c} : \mathfrak{t}}$
$\mathbf{WfObj} \quad \frac{h(\mathfrak{o}) = (C, \tilde{f} : \tilde{v}) \quad \text{fields}(C) = \tilde{f} \tilde{t} \quad \Gamma; h \vdash v_i : t_i}{\Gamma; h \vdash \mathfrak{o}}$		
$\mathbf{WfHeap} \quad \frac{\forall \mathfrak{o} \in \mathcal{D}(h) : \Gamma; h \vdash \mathfrak{o} \quad \forall \mathfrak{o} \in \mathcal{D}(\Gamma) : \Gamma; h \vdash \mathfrak{o} : \Gamma(\mathfrak{o}) \quad \forall \mathfrak{c} \in \mathcal{D}(\Sigma) : \mathfrak{c} \in h \quad \mathcal{D}(\Gamma) \cap \mathcal{D}(\Sigma) = \emptyset}{\Gamma; \Sigma \vdash h}$		

Fig. 21. Types of Runtime Entities, and Well-formed Heaps

Note that $\uparrow \|\theta = \theta\| \uparrow = \perp$.

Using the operator $\|\cdot\|$ the typing rules for processes are straightforward (see Fig. 20). Rule **Start** promotes an expression to the thread level; and rule **Par** types a composition of threads if the composition of their session environments is defined.

In writing session environments we assume the following operator precedence: “,” “ \circ ”, “ $\|\cdot\|$ ”. For example $\Sigma_0, \mathfrak{c} : \pi \circ \Sigma_1 \|\Sigma_2$ is short for $((\Sigma_0, \mathfrak{c} : \pi) \circ \Sigma_1) \|\Sigma_2$.

The typing rules of $\text{MOOSE}_{<}$ are not syntax directed, because of the non structural typing rules and also because of the use of the “ \circ ” and “ $\|\cdot\|$ ” operators in composing session environments. Nevertheless, we can design a type inference algorithm for $\text{MOOSE}_{<}$: as we did for MOOSE [12].

5.3 Subject Reduction

We will consider only reductions of well-typed expressions and threads. We define agreement between environments and heaps in the standard way and we denote it by $\Gamma; \Sigma \vdash h$. The judgement is defined in Fig. 21. The judgement $\Gamma; h \vdash v : t$ guarantees that the value v has type t . The judgement $\Gamma; h \vdash \mathfrak{o}$ guarantees that the object \mathfrak{o} is well-formed, *i.e.* that its fields contain values according to the declared field types in C , the class of that object. The judgement $\Gamma; \Sigma \vdash h$ guarantees that the heap is well-formed for Γ and Σ , *i.e.* that all objects are well-formed, all \mathfrak{o} in the domain of Γ denote in the h objects of the class given to them in Γ , all channels in the domain of Σ are channels in h , and no channel occurs both in Γ and Σ .

We define $\Delta; \Gamma; \Sigma \vdash P; h$ as a shorthand for $\Delta; \Gamma; \Sigma \vdash P$: thread and $\Gamma; \Sigma \vdash h$.

In the following we outline the proof of subject reduction, while we give full details and proofs in the Appendix.

Standard ingredients of Subject Reduction proofs are Generation Lemmas. The Generation Lemmas in this work are somewhat unusual, because, due to the non-structural rules, when an expression is typed, we only can deduce *some* information about the session environment used in the typing. For example, $\Gamma; \Sigma \vdash x : t$ does *not* imply that $\Sigma = \emptyset$; instead, it implies that $\mathcal{R}(\Sigma) \subseteq \{\varepsilon, \varepsilon.\text{end}\}$, where $\mathcal{R}(\Sigma)$ is the range of Σ .

In order to express the Generation Lemmas, we define the partial order \preceq among session environments, which basically reflects the differences introduced through the application of non-structural rules.

Definition 5.2 (Weakening Order \preceq). $\Sigma \preceq \Sigma'$ is the smallest partial order such that:

- $\Sigma \preceq \Sigma, u : \varepsilon$ if $u \notin \mathcal{D}(\Sigma)$,
- $\Sigma, u : \pi \preceq \Sigma, u : \pi.\text{end}$,
- $\Sigma, u : \varepsilon.\text{end} \preceq \Sigma, u : \uparrow$.

The following lemma states that the ordering relation \preceq preserves the types of expressions and threads, and its proof is easy using the non-structural typing rules and Generation Lemmas.

Lemma 5.3. 1. If $\Sigma \preceq \Sigma'$ and $\Delta; \Gamma; \Sigma \vdash e : t$, then $\Delta; \Gamma; \Sigma' \vdash e : t$.
2. If $\Sigma \preceq \Sigma'$ and $\Delta; \Gamma; \Sigma \vdash P : \text{thread}$, then $\Delta; \Gamma; \Sigma' \vdash P : \text{thread}$.

Using the above lemma and the Generation Lemmas one can show that the structural equivalence preserves typing.

Lemma 5.4 (Preservation of Typing under Structural Equivalence). If $\Delta; \Gamma; \Sigma \vdash P : \text{thread}$ and $P \equiv P'$, then $\Delta; \Gamma; \Sigma \vdash P' : \text{thread}$.

Lemma 5.5 states that the typing of $E[e]$ can be broken down into the typing of e , and the typing of $E[x]$. Furthermore, Σ , the environment used to type $E[x]$, can be broken down into two environments, $\Sigma = \Sigma_1 \circ \Sigma_2$, where Σ_1 is used to type e , and Σ_2 is used to type $E[x]$.

Lemma 5.5 (Subderivations). If $\Delta; \Gamma; \Sigma \vdash E[e] : t$, then there exist $\Sigma_1, \Sigma_2, t', x$ fresh in E, Γ , such that $\Sigma = \Sigma_1 \circ \Sigma_2$, and $\Delta; \Gamma; \Sigma_1 \vdash e : t'$, and $\Delta; \Gamma; x : t'; \Sigma_2 \vdash E[x] : t$.

On the other hand, Lemma 5.6 allows the combination of the typing of $E[x]$ and the typing of e , provided that the contexts Σ_1 and Σ_2 used for the two typing can be composed through \circ , and that the type of e is the same as the one of x in the first typing.

Lemma 5.6 (Context Substitution). If $\Delta; \Gamma; \Sigma_1 \vdash e : t'$, and $\Delta; \Gamma; x : t'; \Sigma_2 \vdash E[x] : t$, and $\Sigma_1 \circ \Sigma_2$ is defined, then $\Delta; \Gamma; \Sigma_1 \circ \Sigma_2 \vdash E[e] : t$.

We can now state the Subject Reduction theorem:

Theorem 5.7 (Subject Reduction)

1. $\Delta; \Gamma; \Sigma \vdash e : t$, and $\Gamma; \Sigma \vdash h$, and $e, h \longrightarrow e', h'$ imply $\Delta; \Gamma'; \Sigma \vdash e' : t$, and $\Gamma'; \Sigma \vdash h'$, with $\Gamma \subseteq \Gamma'$.
2. $\Delta; \Gamma; \Sigma \vdash P; h$ and $P, h \longrightarrow P', h'$ imply $\Delta; \Gamma'; \Sigma' \vdash P; h'$ with $\Gamma \subseteq \Gamma'$ and $\Sigma \subseteq \Sigma'$.

The proof, given in the Appendix, is by structural induction on the derivation $e, h \longrightarrow e', h'$ or $P, h \longrightarrow P', h'$. It uses the Generation Lemmas, the Subderivations Lemma, and the Context Substitution Lemma, as well further lemmas, stated and proven in the Appendix, and which deal with properties of the relation “ \succeq ”, of the operators “ \circ ”, “ \parallel ”, and of substitutions.

6 Conclusion and Further Work

In this paper we presented the language $\text{MOOSE}_{<}$, through which we studied the addition of bounded polymorphism to an object-oriented language with session types, and the selection of communication based on the class of objects being sent/received. Through a case study we demonstrated how these new features add flexibility and expressibility. Because of covariance/contravariance of output/input, and the need to match cases, $\text{MOOSE}_{<}$: subtype and duality relations are more interesting than in most work on session types.

In terms of the type systems, there are several directions we plan to explore: We want to extend $\text{MOOSE}_{<}$ to incorporate generic classes in the style of GJ [22], and allow method generic parameters to appear within class instantiations in the argument and result types. We would also like to add union types [26] so as to require the class of objects sent in case expressions to be one of the expected classes and thus guarantee applicability of rule **ComSCaseSuccess-R**. Finally, more refined notions of polymorphism, such as F-bounded polymorphism [4] and match-bounded polymorphism [3], deserve investigations in the framework of session types for object-oriented languages.

In terms of the language design, we believe that selection of communication based on the class of objects being sent/received, rather than on the basis of a label or a boolean, is the right design choice in the context of an object oriented language. However, so far, we have only considered what is a natural *extension* of an object oriented language. A more interesting question to tackle in the future, is an *amalgamation* of object orientation with session primitives.

References

1. Bonelli, E., Compagnoni, A., Gunter, E.: Correspondence Assertions for Process Synchronization in Concurrent Communications. *Journal of Functional Programming* 15(2), 219–248 (2005)
2. Bracha, G., Odersky, M., Stoutamire, D., Wadler, P.: Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In: Chambers, C. (ed.) *OOPSLA '98*, pp. 183–200. ACM Press, New York (1998)

3. Bruce, K.B., Schuett, A., van Gent, R., Fiech, A.: PolyTOIL: A Type-safe Polymorphic Object-oriented Language. *ACM TOPLAS* 25(2), 225–290 (2003)
4. Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.C.: F-bounded Polymorphism for Object-Oriented Programming. In: *FPCA'89*, pp. 273–280. ACM Press, New York (1989)
5. Carbone, M., Honda, K., Yoshida, N.: A Calculus of Global Interaction Based on Session Types. In: *DCM'06. ENTCS*, Elsevier, Amsterdam (to appear, 2007)
6. Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centred Programming for Web Services. In: De Nicola, R. (ed.) *ESOP'07. LNCS*, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)
7. Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys* 17(4), 471–522 (1985)
8. Castagna, G., De Nicola, R., Varacca, D.: Semantic Subtyping for the π -calculus. In: Panangaden, P. (ed.) *LICS '05*, pp. 92–101. IEEE Computer Society Press, Los Alamitos (2005)
9. Castagna, G., Frisch, A.: A Gentle Introduction to Semantic Subtyping. In: Barahona, P., Felty, A. (eds.) *PPDP'05*, pp. 198–208. ACM Press, New York (2005) (joint ICALP-PPDP keynote talk)
10. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N.: Asynchronous Session Types and Progress for Object-Oriented Languages. In: Bonsangue, M., Johnsen, E.B. (eds.) *FMOODS 2007. LNCS*, vol. 4468, pp. 1–31. Springer, Heidelberg (2007)
11. Corin, R., Denielou, P.-M., Fournet, C., Bhargavan, K., Leifer, J.: Secure Implementations for Typed Session Abstractions. In: *CFS'07, IEEE-CS Press*, Los Alamitos (to appear, 2007)
12. Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Drossopoulou, S.: Session Types for Object-Oriented Languages. In: Thomas, D. (ed.) *ECOOP 2006. LNCS*, vol. 4067, pp. 328–352. Springer, Heidelberg (2006)
13. Dezani-Ciancaglini, M., Yoshida, N., Ahern, A., Drossopoulou, S.: A Distributed Object Oriented Language with Session Types. In: De Nicola, R., Sangiorgi, D. (eds.) *TGC 2005. LNCS*, vol. 3705, pp. 299–318. Springer, Heidelberg (2005)
14. Frisch, A., Castagna, G., Benzaken, V.: Semantic Subtyping. In: Plotkin, G. (ed.) *LICS'02*, pp. 137–146. IEEE Computer Society Press, Los Alamitos (2002)
15. Garralda, P., Compagnoni, A., Dezani-Ciancaglini, M.: BASS: Boxed Ambients with Safe Sessions. In: Maher, M. (ed.) *PPDP'06*, pp. 61–72. ACM Press, New York (2006)
16. Gay, S., Hole, M.: Subtyping for Session Types in the Pi-Calculus. *Acta Informatica* 42(2/3), 191–225 (2005)
17. Gay, S., Vasconcelos, V.T., Ravara, A.: Session Types for Inter-Process Communication. TR 2003–133, Department of Computing, University of Glasgow (2003)
18. Gay, S.J.: Bounded Polymorphism in Session Types. *Mathematical Structures in Computer Science* (to appear, 2007)
19. Honda, K.: Types for Dyadic Interaction. In: Best, E. (ed.) *CONCUR 1993. LNCS*, vol. 715, pp. 509–523. Springer, Heidelberg (1993)
20. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Disciplines for Structured Communication-based Programming. In: Hankin, C. (ed.) *ESOP 1998 and ETAPS 1998. LNCS*, vol. 1381, pp. 22–138. Springer, Heidelberg (1998)
21. Honda, K., Yoshida, N., Carbone, M.: Web Services, Mobile Processes and Types. *EATCS Bulletin* 2, 160–185 (2007)
22. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a Minimal Core Calculus for Java and GJ. *ACM TOPLAS* 23(3), 396–450 (2001)

23. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, Parts I and II. Information and Computation, 100(1) (1992)
24. Mostrous, D., Yoshida, N.: Two session typing systems for higher-order mobile processes. In: Ronchi Della Rocca, S. (ed.) TLCA 2007. LNCS, vol. 4583, Springer, Heidelberg (2007)
25. Odersky, M., Wadler, P.: Pizza into Java: Translating Theory into Practice. In: Felleisen, M. (ed.) POPL'97, pp. 146–159. ACM Press, New York (1997)
26. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
27. Sparkes, S.: Conversation with Steve Ross-Talbot. ACM Queue 4(2), 14–23 (2006)
28. Takeuchi, K., Honda, K., Kubo, M.: An Interaction-based Language and its Typing System. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994)
29. Vallecillo, A., Vasconcelos, V.T., Ravara, A.: Typing the Behavior of Objects and Components using Session Types. In: Brogi, A., Jacquet, J.-M. (eds.) FOCLASA'02. ENTCS, vol. 68(3), pp. 439–456. Elsevier, Amsterdam (2002)
30. Vasconcelos, V.T., Gay, S., Ravara, A.: Typechecking a Multithreaded Functional Language with Session Types. Theoretical Computer Science 368(1-2), 64–87 (2006)
31. Walker, D.: Substructural Type Systems. In: Pierce, B.C. (ed.) Advanced Topics in Types and Programming Languages, pp. 3–44. MIT Press, Cambridge (2005)
32. Web Services Choreography Working Group: Web Services Choreography Description Language, <http://www.w3.org/2002/ws/chor/>

A Proof of Subject Reduction

A.1 Generation Lemmas

Lemma A.1 (Generation for Standard Expressions)

1. $\Delta; \Gamma; \Sigma \vdash x : t$ implies $\emptyset \preceq \Sigma$ and $x : t' \in \Gamma$ for some t' such that $\Delta \vdash t' <: t$.
2. $\Delta; \Gamma; \Sigma \vdash c : t$ implies $\emptyset \preceq \Sigma$ and $t = s$.
3. $\Delta; \Gamma; \Sigma \vdash \text{null} : t$ implies $\emptyset \preceq \Sigma$.
4. $\Delta; \Gamma; \Sigma \vdash o : t$ implies $\emptyset \preceq \Sigma$ and $o : C \in \Gamma$ for some C such that $\Delta \vdash C <: t$.
5. $\Delta; \Gamma; \Sigma \vdash \text{NullExc} : t$ implies $\emptyset \preceq \Sigma$.
6. $\Delta; \Gamma; \Sigma \vdash \text{this} : t$ implies $\emptyset \preceq \Sigma$ and $\text{this} : C \in \Gamma$ for some C such that $\Delta \vdash C <: t$.
7. $\Delta; \Gamma; \Sigma \vdash e_1; e_2 : t$ implies $\Sigma = \Sigma_1 \circ \Sigma_2$, and $t = t_2$ and $\Gamma; \Sigma_i \vdash e_i : t_i$ for some Σ_i, t_i ($i \in \{1, 2\}$).
8. $\Delta; \Gamma; \Sigma \vdash e.f := e' : t$ implies $\Sigma = \Sigma_1 \circ \Sigma_2$, and $\Gamma; \Sigma_1 \vdash e : C$ and $\Gamma; \Sigma_2 \vdash e' : t$ with $\text{ft} \in \text{fields}(C)$ for some Σ_1, Σ_2, C .
9. $\Delta; \Gamma; \Sigma \vdash e.f : t$ implies $\Gamma; \Sigma \vdash e : C$ and $\text{ft} \in \text{fields}(C)$ for some C .
10. $\Delta; \Gamma; \Sigma \vdash e.m(e_1, \dots, e_n) : t$ implies $\Delta; \Gamma; \Sigma_0 \vdash e : C$, and $\Delta; \Gamma; \Sigma_i \vdash e_i : t_i$ for $1 \leq i \leq n - m$, and $e_{n-m+j} = u_j$ for $1 \leq j \leq m$, and $\Sigma_0 \circ \Sigma_1 \dots \circ \Sigma_{n-m} \circ \{u_1 : \rho_1, \dots, u_m : \rho_m\} \preceq \Sigma$ and $\text{mtype}(m, C) = t_1, \dots, t_{n-m}, \rho_1, \dots, \rho_m \rightarrow t$, for some m ($0 \leq m \leq n$), $\Sigma_i, t_i, u_j, \rho_j, C$ ($1 \leq i \leq n - m$, $1 \leq j \leq m$).
11. $\Delta; \Gamma; \Sigma \vdash \text{new } C : t$ implies $\emptyset \preceq \Sigma$ and $\Delta \vdash C <: t$.
12. $\Delta; \Gamma; \Sigma \vdash \text{new } (s, \bar{s}) : t$ implies $\emptyset \preceq \Sigma$ and $\Delta \vdash (s, \bar{s}) <: t$.
13. $\Delta; \Gamma; \Sigma \vdash \text{spawn } \{e\} : t$ implies $\Sigma' \preceq \Sigma$, and $\text{ended}(\Sigma')$ and $t = \text{Object}$ and $\Delta; \Gamma; \Sigma' \vdash e : t'$ for some Σ', t' .

Proof. By induction on typing derivations, then case analysis over the shape of the expression being typed, and then case analysis over the last rule applied. We just show one paradigmatic case of the inductive step.

(II0) If the expression being typed has the shape $e.m(e_1, \dots, e_n)$, then the last rule applied is **Meth**, or one of the structural rules. We only consider the case where the last applied rule is **Consume**:

$$\frac{\Delta; \Gamma; \Sigma, u : \varepsilon.\text{end} \vdash e.m(e_1, \dots, e_n) : t}{\Delta; \Gamma; \Sigma, u : \uparrow \vdash e.m(e_1, \dots, e_n) : t}$$

By induction hypothesis we get $\Delta; \Gamma; \Sigma_0 \vdash e : C$, and $\Delta; \Gamma; \Sigma_i \vdash e_i : t_i$ for $1 \leq i \leq n - m$, and $e_{n-m+j} = u_j$ for $1 \leq j \leq m$, and

$$\Sigma_0 \circ \Sigma_1 \dots \circ \Sigma_{n-m} \circ \{u_1 : \rho_1, \dots, u_m : \rho_m\} \preceq \Sigma, u : \varepsilon.\text{end}$$

and $\text{mtype}(m, C) = t_1, \dots, t_{n-m}, \rho_1, \dots, \rho_m \rightarrow t$, for some m ($0 \leq m \leq n$), $\Sigma_i, t_i, u_j, \rho_j, C$ ($1 \leq i \leq n - m, 1 \leq j \leq m$). By definition we also have that $\Sigma, u : \varepsilon.\text{end} \preceq \Sigma, u : \uparrow$, and from transitivity of \preceq we obtain that $\Sigma_0 \circ \Sigma_1 \dots \circ \Sigma_{n-m} \circ \{u_1 : \rho_1, \dots, u_m : \rho_m\} \preceq \Sigma, u : \uparrow$.

Lemma A.2 (Generation for Communication Expressions)

1. $\Delta; \Gamma; \Sigma \vdash \text{connect } u \text{ s } \{e\} : t$ implies $s \text{ begin}.\eta$, and $\Delta; \Gamma; \emptyset \vdash u : \text{begin}.\eta$ and $\Delta; \Gamma \setminus u; \Sigma, u : \eta \vdash e : t$, for some η .
2. $\Delta; \Gamma; \Sigma \vdash u.\text{send}(e) : t$ implies $t = \text{Object}$ and $\Delta; \Gamma; \Sigma' \vdash e : t'$ and $\Sigma' \circ \{u : !t\} \preceq \Sigma$ for some Σ', t' .
3. $\Delta; \Gamma; \Sigma \vdash u.\text{receive}(x)\{e\} : t$ implies $\Delta, X <: t'; \Gamma, x : X; \Sigma' \vdash e : t$ and $X \notin \Gamma \cup \Delta \cup \Sigma \setminus u$ and $\{u : ?(X <: t')\} \circ \Sigma' = \Sigma$ for some X, t', Σ' .
4. $\Delta; \Gamma; \Sigma \vdash u.\text{sendS}(u') : t$ implies $t = \text{Object}$ and $\{u' : \eta, u : !(\eta)\} \preceq \Sigma$ for some η .
5. $\Delta; \Gamma; \Sigma \vdash u.\text{receiveS}(x)\{e\} : t$ implies $t = \text{Object}$ and $\Delta, \eta <: X; \Gamma \setminus x; \{x : \eta\} \vdash e : t'$ and $\{u : ?(X <: \eta)\} \preceq \Sigma$ for some X, η .
6. $\Delta; \Gamma; \Sigma \vdash u.\text{sendCase}(e)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\} : t$ implies $\Delta; \Gamma; \Sigma_0 \vdash e : C$ and $\Delta, X_i <: C_i; \Gamma, x : X_i; \Sigma', u : \rho_i \vdash e_i : t \forall i \in \{1, \dots, n\}$, and $\Sigma_0 \circ \Sigma', u : !((C_1.\rho_1, \dots, C_n.\rho_n)) \preceq \Sigma$ for some $C, \Sigma_0, \Sigma', \rho_1, \dots, \rho_n$.
7. $\Delta; \Gamma; \Sigma \vdash u.\text{receiveCase}(x)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\} : t$ implies $\Delta, X_i <: C_i; \Gamma, x : X_i; \Sigma', u : \rho_i \vdash e_i : t \forall i \in \{1, \dots, n\}$, and $\Sigma', u : ?((X_1 <: C_1).\rho_1, \dots, (X_n <: C_n).\rho_n)) \preceq \Sigma$ for some $\Sigma', \rho_1, \dots, \rho_n$.
8. $\Delta; \Gamma; \Sigma \vdash u.\text{sendWhile}(e)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}\{D_1 \triangleright d_1; \dots; D_m \triangleright d_m\} : t$ implies $\Delta; \Gamma; \emptyset \vdash e : C$, and $\Delta; \Gamma; \{u : \pi_i\} \vdash e_i : t$ and $C_i <: C \forall i \in \{1, \dots, n\}$, and $\Delta; \Gamma; \{u : \pi_{n+j}\} \vdash d_j : t$ and $D_j <: C \forall j \in \{1, \dots, m\}$, and $\{u : !((C_1.\pi_1, \dots, C_n.\pi_n)^*, D_1.\pi_{n+1}, \dots, D_m.\pi_{n+m})\} \preceq \Sigma$ for some $C, \pi_1, \dots, \pi_{n+m}$.
9. $\Delta; \Gamma; \Sigma \vdash u.\text{receiveWhile}(x)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}\{D_1 \triangleright d_1; \dots; D_m \triangleright d_m\} : t$ implies $\Delta, X_i <: C_i; \Gamma, x : X_i; \{u : \pi_i\} \vdash e_i : t \forall i \in \{1, \dots, n\}$, and $\Delta, Y_j <: D_j; \Gamma, x : Y_j; \{u : \pi_{n+j}\} \vdash d_j : t \forall j \in \{1, \dots, m\}$, and $\{u : ?(((X_1 <: C_1).\pi_1, \dots, (X_n <: C_n).\pi_n)^*, (Y_1 <: D_1).\pi_{n+1}, \dots, (Y_m <: D_m).\pi_{n+m}))\} \preceq \Sigma$, for some $X_1, \dots, X_n, Y_1, \dots, Y_m, \pi_1, \dots, \pi_{n+m}$.

Proof. Similar to the proof of Lemma [A.1](#).

Lemma A.3 (Generation for Threads)

1. $\Delta; \Gamma; \Sigma \vdash e : \text{thread}$ implies $\Delta; \Gamma; \Sigma \vdash e : t$ for some t .
2. $\Delta; \Gamma; \Sigma \vdash P_1 | P_2 : \text{thread}$ implies $\Sigma = \Sigma_1 \| \Sigma_2$ and $\Delta; \Gamma; \Sigma_i \vdash P_i : \text{thread}$ ($i \in \{1, 2\}$) for some Σ_1, Σ_2 .

Proof. Similar to the proof of Lemma [A.1](#).

A.2 Types Preservation Under Structural Equivalence, and Under Substitutions

As a convenient shorthand, for any two entities x and y which belong to a domain that includes \perp , we use the notation $x \triangleq y$ to indicate that x is defined if and only if y is defined, and if x is defined then $x = y$.

In Lemma [5.4](#) we show that structural equivalence of terms preserves types. To prove this, we first prove in Lemma [A.4](#) the neutrality of element \emptyset , and associativity and commutativity of parallel composition of session environments. Moreover we show in Lemma [A.5](#) various properties of “ \preceq ”, “ $\|$ ”, and “ \circ ” which easily follow from their definitions.

Lemma A.4. 1. $\Sigma_1 \| \emptyset = \Sigma_1 \emptyset \| \Sigma_1$.

2. $\Sigma_1 \| \Sigma_2 \triangleq \Sigma_2 \| \Sigma_1$.

3. $\Sigma_1 \| (\Sigma_2 \| \Sigma_3) \triangleq (\Sigma_1 \| \Sigma_2) \| \Sigma_3$.

Proof. Note that for any Σ, Σ' , if $\Sigma \| \Sigma'$ is defined, then $\mathcal{D}(\Sigma \| \Sigma') = \mathcal{D}(\Sigma) \cup \mathcal{D}(\Sigma')$.

[\(1\)](#) follows from definition of $\|$.

For [\(2\)](#) show $\forall u \in \mathcal{D}(\Sigma_1) \cup \mathcal{D}(\Sigma_2) : \Sigma_1(u) \| \Sigma_2(u) \triangleq \Sigma_2(u) \| \Sigma_1(u)$. For [\(3\)](#) show $\forall u \in \mathcal{D}(\Sigma_1) \cup \mathcal{D}(\Sigma_2) \cup \mathcal{D}(\Sigma_3) : \Sigma_1(u) \| (\Sigma_2(u) \| \Sigma_3(u)) \triangleq (\Sigma_1(u) \| \Sigma_2(u)) \| \Sigma_3(u)$.

The next Lemma, *i.e.* [A.5](#), characterizes small modifications on operations that preserve well-formedness of the session environment compositions, “ $\|$ ” and “ \circ ”, and also the preservation of the relationship “ \preceq ”. It will be used in the proof of Subject Reduction.

We define:

$$\Sigma[u \mapsto \theta](u') = \begin{cases} \theta & \text{if } u = u', \\ \Sigma(u') & \text{otherwise.} \end{cases}$$

A running type is *atomic* if it is of one of the following shapes:

$$\begin{aligned} &?(X <: t), !t, ?(X <: \eta), !(\eta), !(\tilde{C} . \tilde{\rho}), ?((\tilde{X} <: \tilde{C}) . \tilde{\rho}), \\ &!((\tilde{C} . \tilde{\pi})^*, \tilde{C} . \tilde{\pi}), ?((\tilde{X} <: \tilde{C}) . \tilde{\pi})^*, (\tilde{X} <: \tilde{C}) . \tilde{\pi}. \end{aligned}$$

Lemma A.5. 1. $\emptyset \preceq \Sigma_1$, and $\Sigma_1 \| \Sigma_2$ defined imply $\Sigma_2 \preceq \Sigma_1 \| \Sigma_2$.

2. $\Sigma_1 \| \Sigma_2 \preceq \Sigma$, implies that there are Σ'_1, Σ'_2 such that $\Sigma_1 \preceq \Sigma'_1$ and $\Sigma_2 \preceq \Sigma'_2$ and $\Sigma'_1 \| \Sigma'_2 \Sigma$.

3. $\Sigma_1 \preceq \Sigma'_1$, and $\Sigma'_1 \circ \Sigma_2$ defined, imply $\Sigma_1 \circ \Sigma_2$ defined, and $\Sigma_1 \circ \Sigma_2 \preceq \Sigma'_1 \circ \Sigma_2$.

4. *ended*(Σ_1) and $(\Sigma_1 \cup \Sigma'_1) \circ \Sigma_2$ defined imply
 - (a) $\Sigma'_1 \circ \Sigma_2$ defined,
 - (b) $(\Sigma_1 \cup \Sigma'_1) \circ \Sigma_2 \Sigma_1 \parallel \Sigma'_1 \circ \Sigma_2$.
5. $\Sigma \preceq \Sigma'$ implies $\Sigma \setminus \mathbf{u} \preceq \Sigma' \setminus \mathbf{u}$.
6. $\{\mathbf{u} : \tau\} \preceq \Sigma$ implies
 - (a) $\Sigma(\mathbf{u}) \in \{\tau, \tau.\text{end}, \uparrow\}$ and $\mathcal{R}(\Sigma \setminus \mathbf{u}) \subseteq \{\varepsilon, \varepsilon.\text{end}, \uparrow\}$;
 - (b) $\{\mathbf{u} : \tau'\} \preceq \Sigma[\mathbf{u} \mapsto \tau']$ for all τ' .
7. $\Sigma, \mathbf{u} : \rho \preceq \Sigma'$ and $\Sigma' \circ \Sigma''$ defined imply
 - (a) $\Sigma'[\mathbf{u} \mapsto \rho'] \circ \Sigma''$ defined for all ρ' , proviso that ρ' is ended only if ρ is ended;
 - (b) $\Sigma, \mathbf{u} : \rho' \preceq \Sigma'[\mathbf{u} \mapsto \rho']$ for all ρ' .
8. $\Sigma_1 \parallel \Sigma_2$ defined and
 - (a) $\{\mathbf{u} : !t\} \circ \Sigma'_1 \preceq \Sigma_1$ and $\{\mathbf{u} : ?(X <: t')\} \circ \Sigma'_2 \preceq \Sigma_2$ imply $t <: t'$,
 - (b) $\{\mathbf{u} : !\eta\} \circ \Sigma'_1 \preceq \Sigma_1$ and $\{\mathbf{u} : ?(X <: \eta')\} \circ \Sigma'_2 \preceq \Sigma_2$ imply $\eta <: \eta'$.
9. $\Sigma_1 \circ \Sigma_2 \parallel \Sigma_3 \circ \Sigma_4$ defined, and $\Sigma'_1, \mathbf{u} : \rho \preceq \Sigma_1$ and $\Sigma'_3, \mathbf{u} : \rho' \preceq \Sigma_3$, where ρ and ρ' are atomic, imply:
 - (a) $\rho \bowtie \rho'$;
 - (b) $\Sigma_1[\mathbf{u} \mapsto \rho_1] \circ \Sigma_2 \parallel \Sigma_3[\mathbf{u} \mapsto \rho_2] \circ \Sigma_4 = \Sigma_1 \circ \Sigma_2 \parallel \Sigma_3 \circ \Sigma_4$, for all ρ_1, ρ_2 such that $\rho_1 \bowtie \rho_2$ and ρ_1, ρ_2 are ended only if ρ, ρ' are ended too.
10. $\Sigma_1 \circ \Sigma_2 \parallel \Sigma_3 \circ \Sigma_4$ defined, and $\{\mathbf{u} : \pi\} \preceq \Sigma_1$ and $\{\mathbf{u} : \pi'\} \circ \Sigma'_3 = \Sigma_3$ and $\pi \bowtie \pi'$ imply $\Sigma_1[\mathbf{u} \mapsto \varepsilon] \circ \Sigma_2 \parallel \{\mathbf{u} : \varepsilon\} \circ \Sigma'_3 \circ \Sigma_4 = \Sigma_1 \circ \Sigma_2 \parallel \Sigma_3 \circ \Sigma_4$.

Proof. For (1) notice that $\emptyset \preceq \Sigma_1$ implies $\mathcal{R}(\Sigma_1) \subseteq \{\varepsilon, \varepsilon.\text{end}, \uparrow\}$ and that $\Sigma_1 \parallel \Sigma_2$ defined implies $\Sigma_1(\mathbf{u}) \bowtie \Sigma_2(\mathbf{u})$ for all $\mathbf{u} \in \mathcal{D}(\Sigma_1) \cap \mathcal{D}(\Sigma_2)$.

For (2) one can obtain Σ'_1 and Σ'_2 by applying to Σ_1 and Σ_2 the same transformations which build Σ from $\Sigma_1 \parallel \Sigma_2$.

(3) follows easily from the definitions of “ \preceq ” and of “ \circ ”.

(4a) is immediate. For (4b), *ended*(Σ_1) and $(\Sigma_1 \cup \Sigma'_1) \circ \Sigma_2$ defined imply that $\mathcal{D}(\Sigma_1) \cap \mathcal{D}(\Sigma_2) = \emptyset$.

(5) follows from the definition of “ \preceq ”.

(6a) follows from the definition of “ \preceq ” and (6b) is a consequences of (6a).

(6a) implies (7a) and (7b).

The definition of “ \parallel ”, (5) and (6a) imply (8a), (8b) and (9a). Points (9b) and (10) follow from the observation that in all the equated session environments the predicates of \mathbf{u} are \uparrow .

Lemma 5.4 (Preservation of Typing under Structural Equivalence).

If $\Delta; \Gamma; \Sigma \vdash P : \text{thread}$ and $P \equiv P'$ then $\Delta; \Gamma; \Sigma \vdash P' : \text{thread}$.

Proof. By induction on the derivation of \equiv .

For the case where $P' = P \mid \text{null}$, we use Lemma A.3(2), and obtain $\Sigma = \Sigma_1 \parallel \Sigma_2$ and $\Delta; \Gamma; \Sigma_1 \vdash P : \text{thread}$ and $\Delta; \Gamma; \Sigma_2 \vdash \text{null} : \text{thread}$. Using Lemma A.3(1) and Lemma A.1(3) we get $\Delta; \Gamma; \Sigma_2 \vdash \text{null} : t_2$, and $\emptyset \preceq \Sigma_2$. Using Lemma A.5(1), we obtain that $\Sigma_1 \preceq \Sigma$, and from that, with Lemma 5.3(2) we obtain that $\Delta; \Gamma; \Sigma \vdash P : \text{thread}$.

For the other two basic cases use Lemmas A.3(1) and A.4(2)-(3). For the induction case use Lemma A.3(1) and induction hypothesis.

The next goal is to prove that term substitution preserves types (Lemma [A.6](#)).

Lemma A.6 (Preservation of Typing under Substitution)

1. If $\Delta; \Gamma \setminus u; \Sigma \vdash e : t$ and c is fresh then $\Delta; \Gamma; \Sigma[c/u] \vdash e[c/u] : t$.
2. If $\Delta; \Gamma, \text{this} : C; \Sigma \vdash e : t$ and $\Delta; \Gamma; \emptyset \vdash o : C$ then $\Delta; \Gamma; \Sigma \vdash e[o/\text{this}] : t$.
3. If $\Delta, X <: t'; \Gamma, x : X; \Sigma \vdash e : t$ and $\Delta; \Gamma; \emptyset \vdash v : t''$ and $\Delta \vdash t'' <: t'$, then $\Delta; \Gamma; \Sigma[t''/X] \vdash e[v/x] : t$.
4. If $\Delta, X <: \eta; \Gamma; \{x : X\} \vdash e : t$ and c is fresh and $\Delta \vdash \eta' <: \eta$, then $\Delta; \Gamma; \{c : \eta'\} \vdash e[c/x] : t$.

Proof. All points are proven by induction on derivations.

A.3 Types in Subderivations, and Substitutions Within Contexts

Lemma 5.5 (Subderivations). *If $\Delta; \Gamma; \Sigma \vdash E[e] : t$ then there exist $\Sigma_1, \Sigma_2, x, t', x$ fresh in E, Γ , such that $\Sigma = \Sigma_1 \circ \Sigma_2$, and $\Delta; \Gamma; \Sigma_1 \vdash e : t'$, and $\Delta; \Gamma, x : t'; \Sigma_2 \vdash E[x] : t$.*

Proof. By induction on E , and using Generation Lemmas. For example if $E = []; e'$, then $\Delta; \Gamma; \Sigma \vdash e; e' : t$ implies $\Sigma = \Sigma_1 \circ \Sigma_2$ and $\Gamma; \Sigma_1; \pi \vdash e : t'$ and $\Gamma; \Sigma_2; \pi \vdash e' : t$ by Lemma [A.1\(9\)](#). Then we get $\Delta; \Gamma, x : t'; \Sigma_2 \vdash x; e' : t$ by rules **Var** and **Seq**.

Lemma 5.6 (Context Substitution). *If $\Delta; \Gamma; \Sigma_1 \vdash e : t'$, and $\Delta; \Gamma, x : t'; \Sigma_2 \vdash E[x] : t$, and $\Sigma_1 \circ \Sigma_2$ is defined, then $\Delta; \Gamma; \Sigma_1 \circ \Sigma_2 \vdash E[e] : t$.*

Proof. By induction on E , and using the Generation Lemmas.

Theorem 5.7 (Subject Reduction)

1. $\Delta; \Gamma; \Sigma \vdash e : t$, and $\Gamma; \Sigma \vdash h$, and $e, h \longrightarrow e', h'$ imply $\Delta; \Gamma'; \Sigma \vdash e' : t$, and $\Gamma'; \Sigma \vdash h'$, with $\Gamma \subseteq \Gamma'$.
2. $\Delta; \Gamma; \Sigma \vdash P; h$ and $P, h \longrightarrow P', h'$ imply $\Delta; \Gamma'; \Sigma' \vdash P; h'$ with $\Gamma \subseteq \Gamma'$ and $\Sigma \subseteq \Sigma'$.

Proof. By induction on the reduction $e, h \longrightarrow e', h'$. We only consider the most interesting cases.

Rule Spawn-R

Therefore, the expression being reduced has the form $E[\text{spawn} \{ e \}]$, and

- 0) $h' = h$ and $P' = E[\text{null}] | e$,
- 1) $\Delta; \Gamma; \Sigma \vdash E[\text{spawn} \{ e \}] : t$,
- 2) $\Gamma; \Sigma \vdash h$.

The aim of the next steps is to derive types for e and for $E[\text{null}]$.

Applying Lemma [5.5](#) on [1\)](#) we obtain for some t', Σ_1, Σ_2 :

- 3) $\Delta; \Gamma; \Sigma_1 \vdash \text{spawn} \{ e \} : t'$,
- 4) $\Sigma = \Sigma_1 \circ \Sigma_2$,

5) $\Delta; \Gamma, x : t'; \Sigma_2 \vdash E[x] : t$.

From [3](#)) and Lemma [A.1\(13\)](#), we get for some t'', Σ'_1 :

6) $t' = \text{Object}$,

7) $\Delta; \Gamma; \Sigma'_1 \vdash e : t''$,

8) $\text{ended}(\Sigma'_1)$,

9) $\Sigma'_1 \preceq \Sigma_1$.

From [5](#)), type rule **Null**, and Lemma [5.6](#), we get:

10) $\Delta; \Gamma; \Sigma_2 \vdash E[\text{null}] : t$.

From [10](#)) and rule **Start**, and from [7](#)) and rule **Start**, we obtain:

11) $\Delta; \Gamma; \Sigma_2 \vdash E[\text{null}] : \text{thread}$,

12) $\Delta; \Gamma; \Sigma'_1 \vdash e : \text{thread}$.

From [11](#)), [12](#)) and rule **Par** we get:

13) $\Delta; \Gamma; \Sigma'_1 \parallel \Sigma_2 \vdash e \mid E[\text{null}] : \text{thread}$.

The aim of the next steps is to derive types for $e \mid E[\text{null}]$ in the session environment Σ .

From [4](#)) we obtain that $\Sigma_1 \circ \Sigma_2$ is defined, and therefore, from [9](#)) and Lemma [A.5\(3\)](#), we obtain:

14) $\Sigma'_1 \circ \Sigma_2$ is defined, and $\Sigma'_1 \circ \Sigma_2 \preceq \Sigma_1 \circ \Sigma_2$.

Also, from [8](#)), Lemma [A.5\(4b\)](#), we obtain:

15) $\Sigma'_1 \parallel \Sigma_2 = \Sigma'_1 \circ \Sigma_2$.

Therefore, from [13](#)), [14](#)), [15](#)), and Lemma [5.3\(2\)](#), we obtain:

16) $\Delta; \Gamma; \Sigma \vdash e \mid E[\text{null}] : \text{thread}$.

The case concludes by taking $\Sigma' = \Sigma$, $\Gamma' = \Gamma$ and with [15](#)) and [16](#)).

Rule Connect-R

Then, we have that

0) $P = E_1[\text{connect } c \text{ s } \{e_1\}] \mid E_2[\text{connect } c \text{ s}' \{e_2\}]$,

1) $h' = h :: c'$, with c' is fresh in h ,

2) $P' = E_1[e_1[c'/c]] \mid E_2[e_2[c'/c]]$.

The aim of the next steps is to derive types for e_1 and for e_2 .

From premises, [0](#)) and Lemmas [A.3\(2\)](#), and [A.3\(1\)](#) we obtain for some $\Sigma_1, \Sigma_2, t_1, t_2$:

3) $\Sigma = \Sigma_1 \parallel \Sigma_2$,

4) $\Delta; \Gamma; \Sigma_i \vdash E_i[\text{connect } c \text{ s}_i \{e_i\}] : t_i$ for $(i \in \{1, 2\})$,

5) $\Gamma; \Sigma \vdash h$,

where $s_1 \bowtie s_2$.

From [4](#)), applying Lemma [5.5](#), there exist $\Sigma_{11}, \Sigma_{12}, \Sigma_{21}, \Sigma_{22}, t'_1, t'_2$, such that:

6) $\Sigma_i = \Sigma_{i1} \circ \Sigma_{i2}$,

7) $\Delta; \Gamma; \Sigma_{i1} \vdash \text{connect } c \text{ s}_i \{e_i\} : t'_i$ ($i \in \{1, 2\}$),

8) $\Delta; \Gamma, x_i : t'_i; \Sigma_{i2} \vdash E_i[x_i] : t_i$ ($i \in \{1, 2\}$).

From [7](#)), and Lemma [A.2\(1\)](#) we obtain for some η_1, η_2 :

9) $\Delta; \Gamma; \emptyset \vdash c : s_i$,

10) $s_i = \text{begin}.\eta_i$,

11) $\Delta; \Gamma \setminus c; \Sigma_{i1}, c : \eta_i \vdash e_i : t'_i \ (i \in \{1, 2\})$.

The aim of the next steps is to derive types for P' in a session environment Σ' , so that $\Sigma \subseteq \Sigma'$.

From [\(1\)](#) and [\(11\)](#), and Lemma [A.6\(11\)](#), we get:

12) $\Delta; \Gamma; \Sigma_{i1}, c' : \eta_i \vdash e_i[c'/c] : t'_i \ (i \in \{1, 2\})$.

From [\(12\)](#), [\(8\)](#) and Lemma [5.6](#), we obtain (notice that $(\Sigma_{i1}, c' : \eta_i) \circ \Sigma_{i2}$ is defined by [\(6\)](#) since c' is fresh):

13) $\Delta; \Gamma; (\Sigma_{i1}, c' : \eta_i) \circ \Sigma_{i2} \vdash E_i[e_i[c'/c]] : t'_i \ (i \in \{1, 2\})$.

Applying rules **Start** and **Par** on [\(13\)](#), and also the fact that

$$(\Sigma_{11}, c' : \eta_1) \circ \Sigma_{12} \parallel (\Sigma_{21}, c' : \eta_2) \circ \Sigma_{22} = \Sigma, c' : \uparrow,$$

since $s_1 \bowtie s_2$ implies $\eta_1 \bowtie \eta_2$, we obtain:

14) $\Delta; \Gamma; \Sigma, c' : \uparrow \vdash E_1[e_1[c'/c]] \mid E_2[e_2[c'/c]] : \text{thread}$.

Take

15) $\Sigma' = \Sigma, c' : \uparrow$.

This gives, trivially that:

16) $\Sigma \subseteq \Sigma'$.

Also, from [\(1\)](#) and [\(5\)](#) we obtain:

17) $\Gamma; \Sigma' \vdash h'$.

The case concludes by considering [\(14\)](#), [\(15\)](#), [\(16\)](#) and [\(17\)](#).

Rule ComS-R

Therefore, we have that

0) $P = E_1[c.\text{send}(v)] \mid E_2[c.\text{receive}(x)\{e\}]$, $P' = E_1[\text{null}] \mid E_2[e[v/x]]$,

1) $h' = h$, $\Gamma; \Sigma \vdash h$.

From [\(0\)](#), and from the premises, we obtain by Lemma [A.3\(2\)](#) and [A.3\(1\)](#) that for some $\Sigma_1, \Sigma_2, t_1, t_2$:

2) $\Delta; \Gamma; \Sigma_1 \vdash E_1[c.\text{send}(v)] : t_1$,

3) $\Delta; \Gamma; \Sigma_2 \vdash E_2[c.\text{receive}(x)\{e\}] : t_2$,

4) $\Sigma = \Sigma_1 \parallel \Sigma_2$.

The aim of the next steps is to derive types for $c.\text{receive}(x)\{e\}$ and $c.\text{send}(v)$, and for $E_1[x]$ and $E_2[x]$.

From [\(2\)](#) and Lemma [5.5](#), we obtain for some $\Sigma_{11}, \Sigma_{12}, t'_1$:

5) $\Delta; \Gamma; \Sigma_{11} \vdash c.\text{send}(v) : t'_1$,

6) $\Delta; \Gamma, z : t'_1; \Sigma_{12} \vdash E_1[z] : t_1$,

7) $\Sigma_1 = \Sigma_{11} \circ \Sigma_{12}$.

From [\(5\)](#) and Lemmas [A.2\(2\)](#) and [A.1\(2\)](#), [A.1\(3\)](#), [A.1\(4\)](#), we obtain for some t''_1 :

8) $\Delta; \Gamma; \emptyset \vdash v : t''_1$,

9) $\{c : t''_1\} \preceq \Sigma_{11}$.

From [\(3\)](#), and Lemma [5.5](#), we obtain for some $\Sigma_{21}, \Sigma_{22}, t'_2$:

10) $\Delta; \Gamma; \Sigma_{21} \vdash c.\text{receive}(x)\{e\} : t'_2$,

11) $\Delta; \Gamma, y : t'_2; \Sigma_{22} \vdash E_2[y] : t_2$,

$$12) \quad \Sigma_2 = \Sigma_{21} \circ \Sigma_{22}.$$

From [\(10\)](#), by Lemma [A.2\(3\)](#), we obtain for some X, t_2'' and Σ'_{21} :

$$13) \quad \{c : ?(X <: t_2'')\} \circ \Sigma'_{21} = \Sigma_{21},$$

$$14) \quad \Delta, X <: t_2''; \Gamma, x : X; \Sigma'_{21} \vdash e : t_2'.$$

The aim of the next steps is to derive types for $E_1[\text{null}]$ and $E_2[e[\forall/x]]$.

From [\(9\)](#), and [\(7\)](#), and Lemma [A.5\(7a\)](#) and [\(6b\)](#), we obtain:

$$15) \quad \Sigma_{11}[c \mapsto \varepsilon] \circ \Sigma_{12} \text{ is defined,}$$

$$16) \quad \{c : \varepsilon\} \preceq \Sigma_{11}[c \mapsto \varepsilon].$$

By rules **Null**, and **WeakES**, we obtain $\Delta; \Gamma; \{c : \varepsilon\} \vdash \text{null} : t_1'$.

Then, by [\(16\)](#) and Lemma [5.3\(1\)](#) we obtain:

$$17) \quad \Delta; \Gamma; \Sigma_{11}[c \mapsto \varepsilon] \vdash \text{null} : t_1'.$$

From [\(6\)](#), [\(15\)](#), [\(17\)](#), and Lemma [5.6](#), we obtain:

$$18) \quad \Delta; \Gamma; \Sigma_{11}[c \mapsto \varepsilon] \circ \Sigma_{12} \vdash E_1[\text{null}] : t_1.$$

From [\(7\)](#), [\(9\)](#) by Lemma [A.5\(3\)](#) we get:

$$19) \quad \{c : !t_1''\} \circ \Sigma_{12} \preceq \Sigma_1.$$

[\(12\)](#) and [\(13\)](#) imply:

$$20) \quad \{c : ?(X <: t_2'')\} \circ \Sigma'_{21} \circ \Sigma_{22} = \Sigma_2.$$

[\(19\)](#), and [\(20\)](#) imply by Lemma [A.5\(8a\)](#)

$$21) \quad t_1'' <: t_2''.$$

Therefore, with [\(8\)](#) and [\(14\)](#) we obtain by Lemma [A.6\(3\)](#):

$$22) \quad \Delta; \Gamma; \Sigma'_{21} \vdash e[\forall/x] : t_2',$$

and then by [\(11\)](#) and Lemma [5.6](#) and possibly rule **WeakES**:

$$23) \quad \Delta; \Gamma; \{c : \varepsilon\} \circ \Sigma'_{21} \circ \Sigma_{22} \vdash E_2[e[\forall/x]] : t_2.$$

Furthermore, from [\(4\)](#), [\(7\)](#), [\(12\)](#), [\(9\)](#), [\(13\)](#), [\(21\)](#) and Lemma [A.5\(10\)](#), we obtain:

$$24) \quad \Sigma_{11}[c \mapsto \varepsilon] \circ \Sigma_{12} \parallel \{c : \varepsilon\} \circ \Sigma_{21} \circ \Sigma_{22} = \Sigma_{11} \circ \Sigma_{12} \parallel \Sigma_{21} \circ \Sigma_{22} = \Sigma.$$

The case concludes by applying rules **Par** and **Start** to [\(18\)](#) and [\(23\)](#) taking [\(24\)](#) and [\(7\)](#) into account.

Rule ComSS-R

We have:

$$0) \quad P = E_1[c.\text{sendS}(c')] \mid E_2[c.\text{receiveS}(x)\{e\}],$$

$$1) \quad P' = E_1[\text{null}] \mid e[c/x] \mid E_2[\text{null}],$$

$$2) \quad h' = h, \quad \Gamma; \Sigma \vdash h,$$

$$3) \quad \Delta; \Gamma; \Sigma \vdash P : \text{thread}.$$

From [\(0\)](#), [\(3\)](#) and using Lemma [A.3\(2\)](#) and [\(11\)](#), we obtain for some $\Sigma_1, \Sigma_2, t_1, t_2$:

$$4) \quad \Delta; \Gamma; \Sigma_1 \vdash E_1[c.\text{sendS}(c')] : t_1,$$

$$5) \quad \Delta; \Gamma; \Sigma_2 \vdash E_2[c.\text{receiveS}(x)\{e\}] : t_2,$$

$$6) \quad \Sigma = \Sigma_1 \parallel \Sigma_2.$$

The aim of the next steps is to derive types for $E_1[\text{null}]$ and $E_2[\text{null}]$.

From [\(4\)](#), Lemma [5.5](#) and Lemma [A.2\(4\)](#) we get for some $\Sigma_{11}, \Sigma_{12}, t_1', \eta \neq \varepsilon.\text{end}$:

$$7) \quad \Delta; \Gamma; \Sigma_{11} \vdash c.\text{sendS}(c') : t_1',$$

$$8) \quad \Delta; \Gamma, y : t_1'; \Sigma_{12} \vdash E_1[y] : t_1,$$

$$9) \quad \Sigma_1 = \Sigma_{11} \circ \Sigma_{12},$$

- 10) $t'_1 = \text{Object}$,
- 11) $\{c : !(\eta), c' : \eta\} \preceq \Sigma_{11}$.
 $\square 11$ and Lemma [A.5\(5\)](#) imply
- 12) $\{c' : \eta\} \preceq \Sigma_{11} \setminus c$,
 which gives by $\eta \neq \varepsilon.\text{end}$ and Lemma [A.5\(6a\)](#), for some Σ'_{11} :
- 13) $\Sigma_{11} = \Sigma'_{11}, c' : \eta$.
 $\square 13$ and [9](#) imply by Lemma [A.5\(4a\)](#)
- 14) $\Sigma'_{11} \circ \Sigma_{12}$ defined.
 $\square 11$ and $\square 13$ imply by Lemma [A.5\(5\)](#)
- 15) $\{c : !(\eta)\} \preceq \Sigma'_{11}$.

Using rules **Null**, **WeakES** we obtain:

- 16) $\Delta; \Gamma; \{c : \varepsilon\} \vdash \text{null} : t'_1$.
 By [15](#), [14](#), and Lemma [A.5\(7a\)](#) and [\(6b\)](#) respectively we have:
- 17) $\Sigma'_{11}[c \mapsto \varepsilon] \circ \Sigma_{12}$ defined,
- 18) $\{c : \varepsilon\} \preceq \Sigma'_{11}[c \mapsto \varepsilon]$.

From [18](#), [16](#), and using Lemma [5.3\(1\)](#) we obtain:

- 19) $\Delta; \Gamma; \Sigma'_{11}[c \mapsto \varepsilon] \vdash \text{null} : t'_1$.
 From [8](#), [19](#), [17](#) and Lemma [5.6](#), we obtain:
- 20) $\Delta; \Gamma; \Sigma'_{11}[c \mapsto \varepsilon] \circ \Sigma_{12} \vdash E_1[\text{null}] : t_1$.

From [5](#), Lemma [5.5](#) and Lemma [A.2\(5\)](#) we get for some $\Sigma_{21}, \Sigma_{22}, t'_2, \eta' \neq \varepsilon.\text{end}$:

- 21) $\Delta; \Gamma; \Sigma_{21} \vdash c.\text{receiveS}(x)\{e\} : t'_2$
- 22) $\Delta; \Gamma, z : t'_2; \Sigma_{22} \vdash E_2[z] : t_2$,
- 23) $\Sigma_2 = \Sigma_{21} \circ \Sigma_{22}$,
- 24) $t'_2 = \text{Object}$,
- 25) $\{c : ?(X <: \eta')\} \preceq \Sigma_{21}$,
- 26) $\Delta, X <: \eta'; \Gamma \setminus x; \{x : X\} \vdash e : t'$.

With a proof similar to that of [20](#) we can show:

- 27) $\Delta; \Gamma; \Sigma_{21}[c \mapsto \varepsilon] \circ \Sigma_{22} \vdash E_2[\text{null}] : t_2$.

The aim of the next steps is to type $e[c/x]$ and show that the type of c used to type $c.\text{sendS}(c')$ is dual to that used to type $c.\text{receiveS}(x)\{e\}$, and that the parallel composition of the session environments used to type $E_1[\text{null}]$, $E_2[\text{null}]$, and $e[c/x]$ is the same as Σ .

- $\square 13$ and [9](#) imply by Lemma [A.5\(4b\)](#)
- 28) $\Sigma_{11} \circ \Sigma_{12} = \{c' : \eta\} \parallel \Sigma'_{11} \circ \Sigma_{12}$.
 $\square 6$, [9](#), [23](#) and [28](#) imply:
- 29) $\Sigma'_{11} \circ \Sigma_{12} \parallel \Sigma_{21} \circ \Sigma_{22}$ defined.
 From [29](#), [15](#), [25](#) by Lemma [A.5\(9a\)](#) we get:
- 30) $!(\eta) \bowtie ?(X <: \eta')$,
 which implies by definition of \bowtie :
- 31) $\eta <: \eta'$.
 From [26](#) and [31](#) using Lemma [A.6\(4\)](#) we obtain:
- 32) $\Delta; \Gamma; \{c' : \eta\} \vdash e[c'/x] : t'$.
 Again from [29](#), [15](#), [25](#) by Lemma [A.5\(9b\)](#) we get:
- 33) $\Sigma'_{11}[c \mapsto \varepsilon] \circ \Sigma_{12} \parallel \Sigma_{21}[c \mapsto \varepsilon] \circ \Sigma_{22} = \Sigma'_{11} \circ \Sigma_{12} \parallel \Sigma_{21} \circ \Sigma_{22}$.

[6], **[9]**, **[28]**, **[23]**, and **[33]** imply:

$$34) \quad \Sigma = \{c' : \eta\} \parallel \Sigma'_{11}[c \mapsto \varepsilon] \circ \Sigma_{12} \parallel \Sigma_{21}[c \mapsto \varepsilon] \circ \Sigma_{22}.$$

The case concludes by applying rules **Par** and **Start** to **[20]**, **[27]**, **[32]** by taking into account **[34]** and **[2]**.

Rule ComSCaseSuccess-R

Then, we have that:

- 0) $P = E_1[c.\text{sendCase}(o)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}] \parallel E_2[c.\text{receiveCase}(x)\{C'_1 \triangleright e'_1; \dots; C'_m \triangleright e'_m\}]$,
- 1) $h' = h$,
- 2) $\Gamma; \Sigma \vdash h$,
- 3) $P' = E_1[e_i] \parallel E_2[e_k[o/x]], h$,
- 4) $h(o) = (C, \dots)$ and $C <: C_i$ and $\forall j < i (C \not<: C_j)$, where $i \in \{1, \dots, n\}$, and $C <: C'_k$ and $\forall l < k (C \not<: C'_l)$, where $k \in \{1, \dots, m\}$.

From premises, **[0]** and Lemma **[A.3(2)]** and **[A.3(1)]** we obtain for some $\Sigma_1, \Sigma_2, t_1, t_2$:

- 5) $\Sigma = \Sigma_1 \parallel \Sigma_2$,
- 6) $\Delta; \Gamma; \Sigma_1 \vdash E_1[c.\text{sendCase}(o)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}]: t_1$,
- 7) $\Delta; \Gamma; \Sigma_2 \vdash E_2[c.\text{receiveCase}(x)\{C'_1 \triangleright e'_1; \dots; C'_m \triangleright e'_m\}]: t_2$.

[4] and **[2]** imply

- 8) $\Gamma(o) = C$.

From **[6]**, **[7]** applying Lemma **[5.5]**, there exist $\Sigma_{11}, \Sigma_{12}, \Sigma_{21}, \Sigma_{22}, t'_1, t'_2$ so that:

- 9) $\Sigma_1 = \Sigma_{11} \circ \Sigma_{12}, \quad \Sigma_2 = \Sigma_{21} \circ \Sigma_{22}$,
- 10) $\Delta; \Gamma; \Sigma_{11} \vdash c.\text{sendCase}(o)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}: t'_1$,
- 11) $\Delta; \Gamma, y : t'_1; \Sigma_{12} \vdash E_1[y]: t_1$,
- 12) $\Delta; \Gamma; \Sigma_{21} \vdash c.\text{receiveCase}(x)\{C'_1 \triangleright e'_1; \dots; C'_m \triangleright e'_m\}: t'_2$,
- 13) $\Delta; \Gamma, z : t'_2; \Sigma_{22} \vdash E_2[z]: t_2$.

The aim of the next steps is to find types for e_i , and $E_1[e_i]$.

From **[10]**, **[8]**, and Lemmas **[A.2(6)]** and **[A.1(4)]**, we obtain for some $\Sigma_0, \Sigma'_{11}, \rho_1, \dots, \rho_n$:

- 14) $\Delta; \Gamma; \Sigma_0 \vdash o : C'$ for some C' such that $C <: C'$,
- 15) $\emptyset \preceq \Sigma_0$,
- 16) $\Delta; \Gamma; \Sigma'_{11}, c : \rho_j \vdash e_j : t'_1 \forall j \in \{1, \dots, n\}$,
- 17) $\Sigma_0 \circ \Sigma'_{11}, c : !(C_1.\rho_1, \dots, C_n.\rho_n) \preceq \Sigma_{11}$.

[15] and **[17]** imply by Lemma **[A.5(3)]** and transitivity of \preceq :

- 18) $\Sigma'_{11}, c : !(C_1.\rho_1, \dots, C_n.\rho_n) \preceq \Sigma_{11}$,
and then by Lemma **[A.5(7b)]** and **[A.5(7a)]** and **[9]**
- 19) $\Sigma'_{11}, c : \rho_i \preceq \Sigma_{11}[c \mapsto \rho_i]$,
- 20) $\Sigma_{11}[c \mapsto \rho_i] \circ \Sigma_{12}$ is defined.

From **[16]** and **[19]** we get by Lemma **[5.3(1)]**:

- 21) $\Delta; \Gamma; \Sigma_{11}[c \mapsto \rho_i] \vdash e_i : t'_1$,
which together with **[11]**, **[20]** implies by Lemma **[5.6]**
- 22) $\Delta; \Gamma; \Sigma_{11}[c \mapsto \rho_i] \circ \Sigma_{12} \vdash E_1[e_i]: t_1$.

The aim of the next steps is to show that the type of c used to type e_i is dual to that used to type e'_k , and to find types for $e'_k[\circ/x]$ and $E_2[e'_k[\circ/x]]$.

From [\(12\)](#), and Lemma [A.2\(7\)](#), we obtain for some $\Sigma'_{21}, t'_2, \rho'_1, \dots, \rho'_m$:

$$23) \quad \Delta, X_l <: C'_l; \Gamma, x : X_l; \Sigma'_{21}, c : \rho'_l \vdash e'_l : t'_2 \forall l \in \{1, \dots, m\},$$

$$24) \quad \Sigma'_{21}, c : ?((X_1 <: C'_1) \cdot \rho'_1, \dots, (X_m <: C'_m) \cdot \rho'_m) \preceq \Sigma_{21}.$$

Because of [\(18\)](#), [\(24\)](#), being $\Sigma_{11} \circ \Sigma_{12} \parallel \Sigma_{21} \circ \Sigma_{22}$ defined, and by Lemma [A.5\(9a\)](#) we obtain that:

$$25) \quad !((C_1 \cdot \rho_1, \dots, C_n \cdot \rho_n) \bowtie ?((X_1 <: C'_1) \cdot \rho'_1, \dots, (X_m <: C'_m) \cdot \rho'_m)).$$

Therefore, by definition of the duality relation:

$$26) \quad \rho_j \bowtie \rho'_l [C_j \curlyvee C_l/X_l] \forall j \in \{1, \dots, n\} \text{ and } \forall l \in \{1, \dots, m\}.$$

[\(4\)](#) and [\(8\)](#) imply by rules **Oid** and **Sub** (notice that $C_i \curlyvee C'_k \neq \perp$ by [\(4\)](#)):

$$27) \quad \Delta; \Gamma; \emptyset \vdash \circ : C_i \curlyvee C'_k.$$

Similarly to previous case we can derive:

$$28) \quad \Sigma_{21}[c \mapsto \rho'_k [C_i \curlyvee C'_k/X_k]] \circ \Sigma_{22} \text{ defined,}$$

$$29) \quad \Delta, X_k <: C'_k; \Gamma, x : X; \Sigma_{21}[c \mapsto \rho'_k] \vdash e'_k : t'_2.$$

Applying Lemma [A.6\(3\)](#) to [\(29\)](#), [\(27\)](#), [\(26\)](#), and [\(4\)](#), taking into account that X_k can occur only in ρ'_k we derive:

$$30) \quad \Delta; \Gamma; \Sigma_{21}[c \mapsto \rho'_k [C_i \curlyvee C'_k/X_k]] \vdash e'_k[\circ/x] : t'_2,$$

which together with [\(13\)](#), [\(28\)](#) implies by Lemma [5.6](#):

$$31) \quad \Delta; \Gamma; \Sigma_{21}[c \mapsto \rho'_k [C_i \curlyvee C'_k/X_k]] \circ \Sigma_{22} \vdash E_2[e'_k[\circ/x]] : t_2.$$

From [\(26\)](#), [\(9\)](#), [\(19\)](#), [\(24\)](#), [\(5\)](#) by Lemma [A.5\(9b\)](#) we obtain that:

$$32) \quad \Sigma_{11}[c \mapsto \rho_k] \circ \Sigma_{12} \parallel \Sigma_{21}[c \mapsto \rho'_k [C_i \curlyvee C'_k/X_k]] \circ \Sigma_{22} = \Sigma_{11} \circ \Sigma_{12} \parallel \Sigma_{21} \circ \Sigma_{22} = \Sigma.$$

The case concludes by applying rules **Par** and **Start** to [\(22\)](#), [\(31\)](#), and taking into account [\(32\)](#) and [\(7\)](#), [\(2\)](#).

Rule ComSWhile-R

Then, we have that:

$$0) \quad P = E_1[c.\text{sendWhile}(e)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}\{D_1 \triangleright d_1; \dots; D_m \triangleright d_m\}] \mid E_2[c.\text{receiveWhile}(x)\{C'_1 \triangleright e'_1; \dots; C'_{n'} \triangleright e'_{n'}\}\{D'_1 \triangleright d'_1; \dots; D'_{m'} \triangleright d'_{m'}\}],$$

$$1) \quad h' = h,$$

$$2) \quad \Gamma; \Sigma \vdash h,$$

$$3) \quad P' = E_1[\hat{e}] \mid E_2[\check{e}],$$

where we are using the shorthands:

$$4) \quad \hat{e} = c.\text{sendCase}(e)\{C_1 \triangleright e_1^h; \dots; C_n \triangleright e_n^h, D_1 \triangleright d_1; \dots; D_m \triangleright d_m\},$$

$$5) \quad \check{e} = c.\text{receiveCase}(x)\{C'_1 \triangleright e_1^b; \dots; C'_{n'} \triangleright e_{n'}^b, D'_1 \triangleright d'_1; \dots; D'_{m'} \triangleright d'_{m'}\},$$

$$6) \quad e_i^h \equiv e_i; c.\text{sendWhile}(e)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}\{D_1 \triangleright d_1; \dots; D_m \triangleright d_m\},$$

$$7) \quad e_k^b = e'_k;$$

$$c.\text{receiveWhile}(x)\{C'_1 \triangleright e'_1; \dots; C'_{n'} \triangleright e'_{n'}\}\{D'_1 \triangleright d'_1; \dots; D'_{m'} \triangleright d'_{m'}\}.$$

From premises, [\(0\)](#) and Lemma [A.3\(2\)](#), and [A.3\(1\)](#) we obtain for some $\Sigma_1, \Sigma_2, t_1, t_2$:

$$8) \quad \Sigma = \Sigma_1 \parallel \Sigma_2,$$

$$9) \quad \Delta; \Gamma; \Sigma_1 \vdash$$

$$E_1[c.\text{sendWhile}(e)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}\{D_1 \triangleright d_1; \dots; D_m \triangleright d_m\}] : t_1,$$

- 10) $\Delta; \Gamma; \Sigma_2 \vdash$
 $E_2[c.\text{receiveWhile}(x)\{C'_1 \triangleright e'_1; \dots; C'_{n'} \triangleright e'_{n'}\}\{D'_1 \triangleright d'_1; \dots; D'_{m'} \triangleright d'_{m'}\}]: t_2.$
 From [9](#), [10](#) applying Lemma [5.5](#), there exist $\Sigma_{11}, \Sigma_{12}, \Sigma_{21}, \Sigma_{22}, t'_1, t'_2$ so that:
- 11) $\Sigma_1 = \Sigma_{11} \circ \Sigma_{12}, \quad \Sigma_2 = \Sigma_{21} \circ \Sigma_{22},$
 12) $\Delta; \Gamma; \Sigma_{11} \vdash$
 $c.\text{sendWhile}(e)\{C_1 \triangleright e_1; \dots; C_n \triangleright e_n\}\{D_1 \triangleright d_1; \dots; D_m \triangleright d_m\}: t'_1,$
 13) $\Delta; \Gamma, y : t'_1; \Sigma_{12} \vdash E_1[y]: t_1,$
 14) $\Delta; \Gamma; \Sigma_{21} \vdash$
 $c.\text{receiveWhile}(x)\{C'_1 \triangleright e'_1; \dots; C'_{n'} \triangleright e'_{n'}\}\{D'_1 \triangleright d'_1; \dots; D'_{m'} \triangleright d'_{m'}\}: t'_2,$
 15) $\Delta; \Gamma, z : t'_2; \Sigma_{22} \vdash E_2[z]: t_2.$

The aim of the next steps is to find types for \hat{e} , and $E_1[\hat{e}]$.

- From [12](#), and Lemma [A.2\(8\)](#), we obtain for some $\pi_1, \dots, \pi_{n+m}, t'_1$:
- 16) $\{c : !((C_1.\pi_1, \dots, C_n.\pi_n)^*, D_1.\pi_{n+1}, \dots, D_m.\pi_{n+m})\} \preceq \Sigma_{11},$
 17) $\Delta; \Gamma; \emptyset \vdash e : C,$
 18) $\Delta; \Gamma; \{u : \pi_i\} \vdash e_i : t'_1$ and $C_i <: C$ for all $i \in \{1, \dots, n\},$
 19) $\Delta; \Gamma; \{u : \pi_{n+j}\} \vdash d_j : t'_1$ and $D_j <: C$ for all $j \in \{1, \dots, m\}.$
 We will be using $\hat{\pi}$ as a shorthand defined as follows:
 20) $\hat{\pi} = !((C_1.\pi_1^b, \dots, C_n.\pi_n^b, D_1.\pi_{n+1}, \dots, D_m.\pi_{n+m}),$ where
 21) $\pi_i^b = \pi_i \cdot !((C_1.\pi_1, \dots, C_n.\pi_n)^*, D_1.\pi_{n+1}, \dots, D_m.\pi_{n+m})$ for $i \in \{1, \dots, n\}.$
 By application of type rules **Null**, **SendCase**, **Seq**, **SendWhile** on [17](#) and [18](#), and using the shorthands [4](#) and [20](#) we obtain:
 22) $\Delta; \Gamma; \{c : \hat{\pi}\} \vdash \hat{e} : t'_1.$
 By application of Lemma [A.5\(6b\)](#) on [16](#) we get that:
 23) $\{c : \hat{\pi}\} \preceq \Sigma_{11}[c \mapsto \hat{\pi}],$
 which together with [22](#) implies by Lemma [5.3\(1\)](#)
 24) $\Delta; \Gamma; \Sigma_{11}[c \mapsto \hat{\pi}] \vdash \hat{e} : t'_1.$
 Applying Lemma [A.5\(7a\)](#) to [16](#) and [11](#) we have that:
 25) $\Sigma_{11}[c \mapsto \hat{\pi}] \circ \Sigma_{12}$ is defined.
 Therefore applying Lemma [5.6](#) on [13](#), [24](#) and [25](#) we obtain:
 26) $\Delta; \Gamma; \Sigma_{11}[c \mapsto \hat{\pi}] \circ \Sigma_{12} \vdash E_1[\hat{e}]: t_1.$

The aim of the next steps is to find types for \check{e} and $E_2[\check{e}]$.

- By arguments similar to those used to get [16](#) and [18](#), we obtain from [14](#)
 for some $\pi'_1, \dots, \pi'_{n'+m'}, t'_2$:
- 27) $?(((X_1 <: C'_1).\pi'_1, \dots, (X_{n'} <: C'_{n'}).\pi'_{n'})^*,$
 $(Y_1 <: D'_1).\pi'_{n'+1}, \dots, (Y_{m'} <: D'_{m'}).\pi'_{n'+m'}) \preceq \Sigma_{21},$
 28) $\Delta; \Gamma; \{u : \pi'_k\} \vdash e'_k : t'_2$ for all $k \in \{1, \dots, n'\},$
 29) $\Delta; \Gamma; \{u : \pi'_{n'+l}\} \vdash d_l : t'_2$ for all $l \in \{1, \dots, m'\}.$
 We use the shorthand
 30) $\check{\pi} = ?(((X_1 <: C'_1).\pi'_1, \dots, (X_{n'} <: C'_{n'}).\pi'_{n'})^b,$
 $(Y_1 <: D'_1).\pi'_{n'+1}, \dots, (Y_{m'} <: D'_{m'}).\pi'_{n'+m'}),$ where
 31) $\pi'_k{}^b = \pi'_k \cdot ?(((X_1 <: C'_1).\pi'_1, \dots, (X_{n'} <: C'_{n'}).\pi'_{n'})^*,$
 $(Y_1 <: D'_1).\pi'_{n'+1}, \dots, (Y_{m'} <: D'_{m'}).\pi'_{n'+m'})$ for $k \in \{1, \dots, n'\}.$
 Then, by arguments similar to those used to get [26](#), we obtain that:
 32) $\Delta; \Gamma; \Sigma_{21}[c \mapsto \check{\pi}] \circ \Sigma_{22} \vdash E_2[\check{e}]: t_2.$

The aim of the next steps is to show that the type of c used to type \hat{e} is dual to that used to type \check{e} , and that the parallel composition of the session environments used to type $E_1[\hat{e}]$ and $E_2[\check{e}]$ is the same as Σ .

Because of [16](#), [27](#), being $\Sigma_{11} \circ \Sigma_{12} \parallel \Sigma_{21} \circ \Sigma_{22}$ defined, and by Lemma [A.5\(9a\)](#) we obtain that:

$$33) \quad !((C_1.\pi_1, \dots, C_n.\pi_n)^*, D_1.\pi_{n+1}, \dots, D_m.\pi_{n+m}) \bowtie \\ ?(((X_1 <: C'_1).\pi'_1, \dots, (X_{n'} <: C'_{n'}).\pi'_{n'})^*, (Y_1 <: D'_1).\pi'_{n'+1}, \dots, (Y_{m'} <: D'_{m'}).\pi'_{n'+m'}),$$

which implies, by definition of the duality relation:

$$34) \quad \hat{\pi} \bowtie \check{\pi}.$$

Therefore, by Lemma [A.5\(9b\)](#) and using [11](#), [16](#), [27](#), [8](#) we obtain that:

$$35) \quad \Sigma_{11}[c \mapsto \hat{\pi}] \circ \Sigma_{12} \parallel \Sigma_{21}[c \mapsto \check{\pi}] \circ \Sigma_{22} = \Sigma_{11} \circ \Sigma_{12} \parallel \Sigma_{21} \circ \Sigma_{22} = \Sigma.$$

The case concludes by applying rules **Par** and **Start** to [26](#), [32](#), and taking into account [35](#) and [7](#), [2](#).

Reflecting on Aspect-Oriented Programming, Metaprogramming, and Adaptive Distributed Monitoring*

Bill Donkervoet and Gul Agha

Open Systems Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign
{donkervo, agha}@uiuc.edu

Abstract. Metaprogramming and computational reflection are two related techniques that allow the programmer to change the semantics of a program in a modular fashion. Although the concepts have been explored by researchers for some time, a form of metaprogramming, namely aspect-oriented programming, is now being used by some practitioners. This paper is an attempt to understand the limitations of different forms of computational reflection in concurrent and distributed computing. It specifically studies the use of aspect-oriented programming and reflective actor libraries, and their relation to full reflection. We choose distributed monitoring as the primary example application because its requirements nicely fit the abilities of the two systems as well as illustrate their limitations.

1 Introduction

Addressing software complexity through modular decomposition is an old idea. Objects provide one mechanism for modularity—namely, support for abstract data types, thus separating the interface from the representation. The notion of objects generalizes to components and supports a functional decomposition of large software systems. However, the functional behavior of a system represents only one ‘aspect’ of this decomposition. Observe that concurrency is common in real-world systems and sequential computation is simply a degenerate case of concurrent computation; therefore, we focus only on concurrent and distributed systems. Concerns in concurrent systems include synchronization, fault-tolerance, scheduling, real-time, coordination, etc. Code to implement these requirements is a major cause of software complexity.

For almost two decades, programming language researchers have explored mechanisms to support a separation of concerns. The goal of separating code to implement functional (or transformational) requirements from code to satisfy other concerns is motivated by the usual advantages of components: namely to

* This research has been supported in part by NSF under grant CNS 05-09321 and by ONR under DoD MURI award N0014-02-1-0715.

simplify the process of building complex software systems and to facilitate reuse of software modules in different contexts. Researchers have proposed a number of techniques to facilitate a separation of design concerns; these include computational reflection [35,29], metaprogramming [10], generative programming [14], aspect-oriented programming [13], filters [2], coordination constraints [22] and circuits [3]. From a theoretical perspective, all these techniques can be understood as forms of metaprogramming although many of them are far more restrictive than (full) computational reflection.

The idea of *metaprogramming* is to allow the manipulation of computational structures containing a representation of a program and its data [35]. With *computational reflection* (or simply *reflection*, a program may inspect and modify itself while running. Reflection and metaprogramming, two related concepts, are tools that not only simplify certain problems but make it possible to address problems requiring dynamic program adaptation. The basis for computational reflection is provided by a foundational concept in the theory of computation as well as computer architecture: namely, that programs are data and may therefore be stored and manipulated as other data. The semantics and architecture of different programming languages and frameworks provide different kinds and degrees of reflection.

On the more static end of the metaprogramming spectrum lies generative programming. *Generative programming* is a form of metaprogramming where additional code is generated based on a high-level specification and the application source code. This generated code is then added to the original source code before interpretation or compilation. *Aspect-oriented programming* may be thought of as a kind of generative programming, enabling code insertion at certain, well-specified points in a program. AOP allows clean, modular specification of both the primary task and other orthogonal aspects, which are then woven together into a single program either at compile-time or runtime.

Our goal is to further the understanding of the semantics of languages and frameworks supporting a separation of design concerns. We focus on two programming concepts that support a separation of concerns: reflection and aspect-oriented programming. Reflection is a powerful programming mechanism with a formal semantics that has been studied in sequential programming languages (e.g., [20,8]) and in concurrent (actor-based) programming languages [5,16].

One way to understand the expressive power of two programming languages is to encode one in the other and show that equivalence relations hold in the translated system. An example of this kind of semantic analysis can be found in [30], which shows that an actor language with remote procedure calls and local synchronization constraints can be translated into a pure actor language (as is done in [27]) without modifying the actor semantics. From a translational perspective, it is straightforward to represent aspect-oriented programming using reflection [9]. Thus, the interesting problem is to understand the limitations of aspect-oriented programming. The approach we take is to pick an example and study its implementation in a reflective actor system and in a programming system supporting aspects in Java.

In order to best understand the strengths and weaknesses of various forms of reflection, we choose an example of a concurrent program that illustrates modularity, orthogonality, and dynamicity. Specifically, we will explore, compare, and contrast these attributes using distributed monitoring. Distributed monitoring of error conditions and constraint violations is a difficult and, as yet, only partially solved problem. Distributed monitoring requires each node (actor) in a distributed system to record and communicate its knowledge of the world, checking for specified conditions. The necessary knowledge may be communicated efficiently by piggybacking state information along with regular messages, thus propagating state knowledge to acquaintances [33].

Although monitoring of statically defined constraints is relatively straightforward, the ability to add and modify such constraints during runtime requires an added level of flexibility. Past solutions have utilized aspect oriented programming to provide insertion of error checks and modification of message format at compile time. We will describe this and other approaches and then discuss the problem of runtime insertion or modification of monitors. It is our conjecture that such flexibility can only be supported in a fully reflective architecture.

The outline of the paper is as follows. Section 2 introduces metaprogramming, leading into a discussion of reflection in Section 3. Aspect-oriented programming is discussed in Section 4. Section 5 addresses reflection's relation to concurrent programming with threads and with actors. Section 6 introduces the case study problem of distributed monitoring and Section 7 describes our implementation. Section 8 surveys related work and the final section concludes with a summary and discussion of open research problems.

2 Metaprogramming

A *metaprogram* is a program that creates or manipulates another program (possibly itself), where the latter program is represented as data. As mentioned earlier, examples of metaprogramming techniques are reflection [35] in which a program can inspect and modify its own behavior, and generative programming, in which the output of a program is the source code for another program [15].

2.1 Meta-architectures

A *meta-architecture* is a representation that not only captures knowledge about a task being performed but also captures knowledge about the performance of the task. Meta-architectures provide access to *metadata*, i.e., access to information about the data. A meta-architecture provides knowledge about a program beyond the information relevant to the application domain. An example of metadata in operating systems is information about a file—such as its creation date, owner, and access information—that is generally stored with the application data.

Access to metadata can enable certain tasks that are not otherwise possible. For example, metadata is useful for governing control-flow in programs as well as for debugging [24]. Every runtime system necessarily maintains some metadata and thus can be thought of as a meta-architecture. The meta-architecture

the program’s metadata should be reflected in modifications to the data itself. Similarly, modifications to metadata about the program should be reflected in the execution of the program. This produces a causal relationship between the data and the metadata.

Full reflection is composed of two properties: introspection and intercession [32]. *Introspection* is the ability of a program entity to view its internal state or representation. *Intercession* allows program entities to modify their representation, thereby changing the program’s behavior. Since introspection is easier to implement than intercession, some languages (including Java) offer only introspection. On the other hand, languages offering intercession also offer introspective capabilities. This is not surprising given that intercession requires a more complex reflective system.

The use of reflection can be illustrated by the following: Consider a robotic arm; a computer maintains data structures that represent the location and position of the arm; these structures are metadata, they simply exist as the program’s internal representation of the arm, whereas the actual data is the arm’s physical position. Modification of the internal data results in an external movement of the arm. Similarly, external manipulation of the arm results in modification of the internal data. Extending this causal relationship beyond program data to runtime data allows similar modification of the computation itself.

Reflection

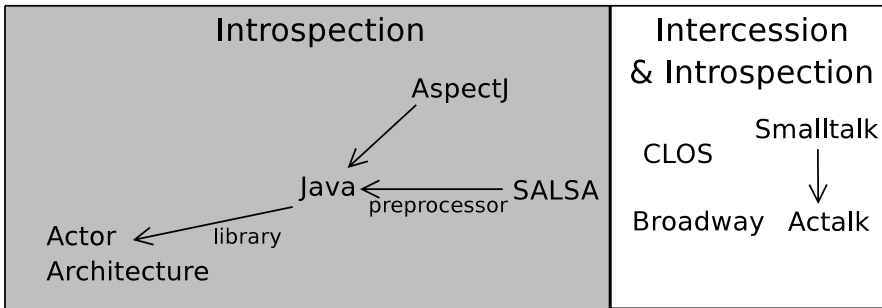


Fig. 1. An illustration of reflective capabilities of various systems. Java and its related systems offer only introspection whereas Smalltalk and CLOS-based systems offer much fuller reflective capabilities.

3.1 Reification

Reification is the process of creating a concrete representation of something abstract. Reification is a necessary step for reflection. In the context of reflection, reification is the act of creating a data representation of the program’s internal data (including current state). Once a reification of the program is created, viewing the reification by the program is reflective introspection. Similarly, if the reification is modifiable by the program, reflective intercession is possible.

3.2 Reflection in the Real World

Since reflection allows modification of the program and how the program is interpreted, full reflection even allows modification of the interpreter and runtime system. Thus, most reflective systems offer only a subset of reflective capabilities. For example, Java reflection primarily allows observation of objects and very little modification [23]. Java objects can determine class information, list and access methods and fields, and create new objects. Fields can be accessed and modified but methods cannot be changed so program modification is not possible. Thus, Java offers introspection but not intercession.

On the other hand, Smalltalk has much more powerful reflective capabilities—allowing both introspection and intercession [19]. Objects can dynamically modify their behaviors, change their methods, fields, and even their class. Since the Smalltalk compiler is part of the Smalltalk library, the entire runtime system can be modified—giving the program almost total flexibility. Even so, there are still reflective facilities not offered by Smalltalk for the sake of efficiency and simplicity: in particular, the limited recursion of metaobjects as described in Section 2.2.

4 Aspect Oriented Programming

Aspect-oriented programming (AOP) is a metaprogramming paradigm that allows separation of concerns through code *cross-cutting*. Different modules of the program are specified, ideally each addressing a singular concern, and these aspects are woven into the main code to produce a single program addressing all concerns. For example, a program can use clean, simple communication method calls and security concerns can later be woven into the program using a security aspect. This not only greatly simplifies the original program but also eases modifications caused by changing specifications; in particular, the original program need not be changed if only some of its aspects change. Similarly, modification of the original program is simplified as it contains only the program logic and not the orthogonal aspect code.

AOP weaves aspects together with the main program at *joinpoints*, points in the program that meet a programmer’s specification. Such points must be at designated “control points” such as entry to method calls, exit from method calls, or object creation or deletion [13]. In the above example of network security, encryption wrappers can be applied to a message at the call joinpoint of the network send method and decryption can take place at the return joinpoint of the network receive method. Although the idea of aspect weaving is very standard, the method can vary substantially and be either very dynamic, using reflection, or completely static, using code generation.

4.1 Code Weaving

One form of AOP weaves aspects into the program code statically in the compilation process as a form of generative programming, which requires no runtime

meta-architecture. This form of aspect weaving can generate woven programs in the original language or in bytecode. This is then run with the aspects compiled directly and permanently into the program as if the aspects had been hardcoded in initially.

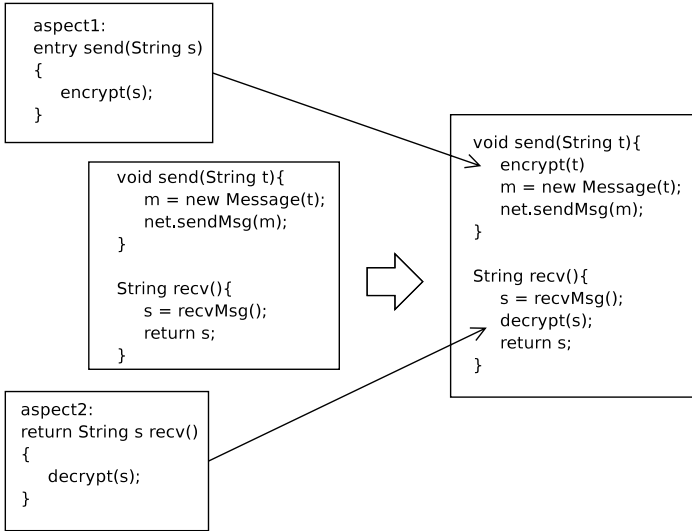


Fig. 2. An example of static code weaving using the secure network example. On the left are two aspects, above and below the regular program code. To the right the code is woven together ready to be compiled and run.

4.2 Reflective AOP

However, if the aspect weaving is done during runtime, certain meta-architectural facilities are necessary. Runtime weaving requires modification of joinpoint behavior and may also require modification of variables.

This behavior modification is achieved through modification of the metadata controlling program execution, although only introspection and a very basic level of self-modification is required. AspectJ, an aspect-oriented language based on Java, utilizes Java's reflective facilities to enable AOP through creating hooks at all specified joinpoints. In the case of a method call joinpoint, the method name and arguments are analyzed to determine if a match occurs and, if so, the aspect code is called before, after, or around the regular method code.

5 Reflection in Concurrency

Because reflection is often a feature of a programming language, some aspects of reflection are unaffected by concurrency. Distributed systems that provide reflection not through a language but through a library often respect the logical

bounds of concurrent structures. Reflection need not change the behavior of the entire system but of individual actors [6] or of a group of actors [42]. In case of actors, formal models of reflection have been developed.

5.1 Threads and Objects

A number of languages that have been designed primarily for sequential computing provide concurrency through threading. One example of such a language is Java. A Java object can be assigned a thread by implementing the `Runnable` interface or subclassing the `Thread` class. However, within that thread and its ‘member’ objects, all communication is synchronous using regular method calls. Because of this, metaobject protocols in Java typically remain unchanged in the presence of concurrency. Because reflection respects the bounds of objects, procedures, and functions just as threads do, concurrency and reflection are non-conflicting. Any reflection is done entirely within the bounds of a single thread; as a consequence, reflection has no effect on the concurrency in the program.

However, in situations where reflection is able to modify the interpreter, compiler, or runtime system, the effect of reflection can drastically affect how the entire system (including threads) behave. For example, the ability to reflectively modify the runtime system means that the scheduler may be modified, influencing thread behavior. Modern operating systems often export some reflective facilities to programs—in particular, this includes access to the scheduler. As a consequence, processes and threads have the ability to modify their priorities and choose the scheduling policy. In some cases, even greater access is granted: processes may modify the scheduler algorithm [28]. Using these system-wide reflective facilities can greatly influence concurrency in multithreaded programs.

5.2 Actors

Actors are a model of concurrent computation that encapsulate not only data and behavior (as objects do) but also have their own locus of control [1]. Actors can send and receive messages, process messages, create new actors, and compute. Because of their simplicity and inherently concurrent nature, they are a natural model for distributed systems.

Just as object oriented languages introduce metaobjects, actor frameworks and languages often provide meta-actors [37]. These are simply the ‘meta’ equivalents of actors. Meta-actors allow access to base-actor internals and behavior. Typically meta-actors only intercept incoming and outgoing messages, delaying, modifying, or discarding them to alter the behavior of the actor system. Such modification of message delivery affects the scheduling of actors, and can be used to modularly provide atomicity or enforce a precedence order in the processing of messages. Another form of reflection in actors is providing the ability to copy state. With this additional reflective capability, arbitrary protocol stacks can be defined [5]. The techniques have been applied to provide separate specification and dynamic composition of dependability protocols (security, fault-tolerance, reliability) with distributed application code [38].

For message interception, the actor send and receive methods are modified so that both incoming and outgoing messages are routed through the meta-actor. The normal operational semantics for sending messages in actors are:

$$\mathbf{send}: [R[\mathit{send}(t, m)], \emptyset]_a \rightarrow [R[\mathit{nil}], \emptyset]_a, < t \Leftarrow m >$$

where R is a reduction context, t is the target, m is the message, a is the local actor, and \emptyset signifies that there is no associated meta-actor.

When a meta-actor, \hat{a} , is installed to intercept messages, the semantics of sending messages are modified so that, rather than sending directly to the message target, the message is passed to the meta-actor using a transmit message.

$$\begin{aligned} \mathbf{send} \text{ (with meta)}: [R[\mathit{send}(t, m)], \hat{a}]_a &\rightarrow [R[\mathit{nil}], \hat{a}]_a, < \hat{a} \Leftarrow (\mathit{transmit}(t, m)) > \\ \mathbf{transmit} \text{ (on meta)}: [R[\mathit{transmit}(t, m)], b]_{\hat{a}} &\rightarrow [R[\mathit{send}(t, m)], b]_{\hat{a}} \end{aligned}$$

The default **transmit** semantics on a meta-actor are simply to propagate the message on to the target. b is either another meta-actor or \emptyset . since meta-actors can be layered, as in the *Russian dolls* model [31], these semantics apply regardless of whether there is another higher level of meta-actor. In the case that $b \neq \emptyset$, \hat{a} 's overloaded **send** method simply passes the message to the next higher meta-actor. The **transmit** semantics can be overridden to achieve other tasks such as modifying the outgoing message.

Similarly, receive semantics are modified. Normal operational semantics of receive allow an actor a in a wait state to receive and apply message m :

$$\mathbf{rcv}: [R[\mathit{wait}()], \emptyset]_a, < a \Leftarrow m > \rightarrow [R[\mathit{app}(m)], \emptyset]_a$$

In a reflective system, the semantics of receiving messages is replaced by two methods that allow interaction with the meta-actor:

$$\begin{aligned} \mathbf{rcv} \text{ (with meta)}: < a \Leftarrow m > \rightarrow < (\hat{a} \Leftarrow \mathit{dlv}(m)) > \\ &\text{where } \mathit{meta}(a) = \hat{a} \\ \mathbf{proc} \text{ (with meta)}: [R[\mathit{wait}()], \hat{a}]_a, < a \Leftarrow m >_{\mathit{meta}} &\rightarrow [R[\mathit{app}(m)], \hat{a}]_a \end{aligned}$$

rcv redirects the incoming message to the meta-actor in **dlv**, a deliver message. **proc** is the second half of the receive, when the message is actually delivered to the actor by its meta-actor. $< a \Leftarrow m >_{\mathit{meta}}$ is a special message transmission from a meta-actor to its associated base. This is required because a message sent using the traditional means would again be redirected to the meta-actor.

The final piece of the actor/meta-actor communication is the meta-actor semantics for a message receive.

$$\begin{aligned} \mathbf{dlv} \text{ (on meta)}: [R[\mathit{dlv}(m)], b]_{\hat{a}} &\rightarrow [R[\mathit{nil}], b]_{\hat{a}}, < a \Leftarrow m >_{\mathit{meta}} \\ &\text{where } \mathit{meta}(a) = \hat{a} \end{aligned}$$

Similarly to **transmit**, the **dlv** message can be overridden to achieve desired meta-actor functionality.

Two-level Actor Model. Although the above semantics allow for any number of meta-actors, as in object-oriented languages, actor systems often limit the number of meta-levels for practical reasons. In the Two-Level Actor Model (TLAM), there is a base-level actor and an optional meta-actor [39]. This two-level, reflective architecture provides a dynamic, flexible distributed system in which the meta-actor may alter or enhance the behavior of the base-actor [40].

For c, t actor ids, n a number

States: $T(n)$ Messages: $tick, time@c, reply(n)$

Reaction Rules:

$$\begin{aligned} (t|T(n)) : < t \Leftarrow tick > \rightarrow (t|T(n+1)) : < t \Leftarrow tick > \\ (t|T(n)) : < t \Leftarrow time@c > \rightarrow (t|T(n)) : < c \Leftarrow reply(n) > \end{aligned}$$

Fig. 3. Tick base-level actors in TLAM

Figures 3 and 4 show an example TLAM system borrowed from [39]. The base-level actors (Figure 3), respond to *tick* messages by incrementing their internal counter and sending another *tick* message to themselves; they respond to *time@c* messages by maintaining their previous state and sending the current counter value to address c . The meta-level actors are used as a logging service and to manipulate a base-level actor by resetting its counter to zero (Figure 4). The first meta-actor rule dictates that on delivering a *time@c* message to its base-actor, the meta-actor logs the event and participants by sending a *log* message to the observer o . On receipt of a *reset* message, the meta-actor resets the value of its base-level actor's counter to zero and sends a *resetAck* message to o .

For t, o, c actor ids, n, m numbers

States: $M(t, o, m)$ Messages: $log(t, n, m, c), reset, resetAck$

Reaction Rules:

$$\begin{aligned} (tm|M(t, o, m)) : dlv((t|T(n)) : < t \Leftarrow time@c >) \rightarrow \\ (tm|M(t, o, m+1)) : < o \Leftarrow log(t, n, m+1, c) > \\ (tm|M(t, o, m)) : < tm \Leftarrow reset > \rightarrow \\ (tm|M(t, o, 0)) : \{ /t := T(0) \}, < o \Leftarrow resetAck > \end{aligned}$$

Fig. 4. Tick Monitor using meta-actors in TLAM

The logging example shows how meta-actors can be used to address secondary concerns in a modular manner. A more involved problem that has been addressed by TLAM is that of garbage collection [40]. Using meta-actors to create a reachability snapshot, unreachable base-level actors can be detected and garbage collected. Meta-actor functionality may be composed with the TLAM migration service to provide garbage collection that works in the presence of migration. Another application of the model has been to provide modular specification and implementation for Quality of Service requirements in multimedia [41].

6 Case Study: Distributed Monitoring

To understand the strengths and weaknesses of various reflective techniques, we choose to study a specific example. The task implemented by reflection should be orthogonal to the functional (transformational) behavior of an application. However, the reflective behavior needs to be related to the primary task in such a way that it must make use of the state or other internals of the primary task. If there were not such a relation, there would be no reason for the primary and reflective tasks to be joined; they would simply be separate programs.

Considering all of these requirements, we've chosen distributed monitoring as a case study. Distributed monitoring involves runtime observation of safety or error conditions of a distributed system. There are several monitoring approaches, each with advantages and disadvantages. Each approach also requires a different level of reflection, thus offering a different level of flexibility and dynamicity.

6.1 Monitoring Details

Distributed monitoring may be done either centrally or in a decentralized manner. Centralized approaches maintain a single monitor to which all distributed nodes report. This provides a global, sequential view of the entire system and makes monitoring extremely simple. However, this also violates the goals of a distributed system by providing a single point of failure and a system bottleneck hindering scalability.

Distributed monitoring of a distributed system requires monitors local to each distributed node. In order to perform the monitoring task, the monitors must each maintain a view of the entire monitored system. Thus, state data for monitoring is propagated with normal messages between nodes.

Because each node maintains recent state information gathered from messages from remote nodes, it may not have the actual current global state. Thus, the distributed monitor is causally correct as it maintains causally consistent data whereas a centralized monitor is able to assure that the monitor is sequentially correct by maintaining sequentially consistent data [25].

Throughout this section we will use the terms static and dynamic; by *static monitors* we refer to the fact that the installation of monitors must be done at compile time. *Dynamic monitors* means that the monitor may be installed or removed during runtime without redeploying the system. Unfortunately, the terms static and dynamic are somewhat overloaded—even a statically installed monitor observes the system at runtime. Moreover, the dynamicity of a monitor also depends on other factors: a monitor may adapt to changes in a distributed system, such as nodes joining and leaving system. Finally, we wish to address the ability of the monitoring system as a whole to adapt and cooperate through inter-monitor messaging.

Sen et al. introduce past-time distributed temporal logic, PT-DTL, as a language for defining monitors to specify restrictions on past or currently-known values at local or remote nodes [34]. Monitor code is generated from the logic requirements that is then woven into the code for deployment.

6.2 Past-Time Distributed Temporal Logic

PT-DTL is a logic for specifying passive monitor conditions using traditional logical and propositional operators. Past-time temporal operators for dictating previous states include **previously**, **always in the past**, **happens-after**, and **at some time in the past**. These sets of operators form past-time linear temporal logic, PT-LTL. In order to address distributed computation, the epistemic operators $@_{\forall J}FJ$, $@_{\exists J}FJ$, and $@_j(\text{some function})$ are added to complete PT-DTL.

For example, the following formula tests for the safety of leader election:

$$@_i(\text{leaderElected} \rightarrow ((\text{state} = \text{leader}) \rightarrow (@_{\{\forall j|j \neq i\}}(\text{state} \neq \text{leader}))))$$

where the beginning of every PT-DTL statement is $@_i$, stating that the following constraint is being monitored at the local node. **leaderElected**, **state**, and **leader** are local keywords in the monitored process. Finally, the $@_{\{\forall j|j \neq i\}} \dots$ is the epistemic component; evaluating based on the local knowledge of the most recently known states of other nodes in the system.

Although PT-DTL is a simple distribution of PT-LTL, its distributed nature means that it has a looser consistency model than PT-LTL or a centralized approach. PT-DTL must maintain not only the current values of all other nodes but also the evaluation of past-time logic expressions in order to maintain history.

In addition to the operators provided by PT-DTL, our implementation also provides a means of communication between monitors, allowing cooperation and synchronization. Using this added ability, monitors may perform model-based monitoring where the global monitoring scheme changes in response to the current system state. The new scheme is communicated via this monitor channel and each monitor adapts accordingly. Thus, we require a small logic extension to PT-DTL, which we hope to address in the near future.

6.3 Example Application

Suppose we have a network of distributed temperature monitors to assure a safe operating environment. We wish to ensure that the temperature at any one monitor is not greater than 110% of the average temperature. The PT-DTL formula for such a monitor would be:

$$@_i(\text{temp} < (1.10 * \text{avg}(@_{\{j|j \text{ is any process}\}}(\text{temp}))))$$

Assuming the datatype holding the knowledge vector is sufficiently flexible to address joining and leaving nodes, this should maintain our operating constraints. However, as simple a change as modifying the alarm threshold to 125% requires flexibility of the monitor itself. In this case, such flexibility could be provided easily enough by using a variable to set the alarm tolerance.

However, using variables, flexible data types, and foresight can only address problems to a point before the amount of flexibility required pushes design requirements into the reflective realm. For example, if instead of monitoring the system as a whole, we wish to monitor each individual room:

$$@_i(\text{temp} < (1.10 * \text{avg}(@_{\{j|i, j \in \text{room}(J)\}}(\text{temp}))))$$

or the global system and each individual room with different tolerances:

$$\begin{aligned} & @_i ((\mathbf{temp} < (1.25 * \mathbf{avg}(@_{\{j|j \text{ is any process}\}}(\mathbf{temp})))) \\ & \wedge (\mathbf{temp} < (1.10 * \mathbf{avg}(@_{\{j|i,j \in \text{room.J}\}}(\mathbf{temp})))) \end{aligned}$$

It quickly becomes evident that a more flexible system is necessary.

Using a computationally reflective system, it would be possible to modify methods instead of just modifying variable values. In the above example, the monitor method could be changed on the fly to accommodate for the changing requirements. Using a non-reflective system, the method change would have to be done in code, recompiled, and then deployed.

6.4 AOP in Distributed Monitoring

Many current distributed monitoring applications use aspect oriented programming to weave the monitor code in with the program code [12,34]. In these systems, monitors are often compiled from a monitor logic into aspects, which are then woven into the primary task's program code in appropriate places.

AOP keeps monitor code orthogonal to program code as stated in our case study requirements. Additionally, aspects have access to program internals such as state and variables allowing program monitoring and can even maintain their own state, which is useful for model-based monitors. Although the AOP approach can address monitoring requirements, it is not as flexible as other reflective approaches. Namely, since aspects must be woven into program code, they cannot be added, modified, or removed at runtime but are set at compile time.

6.5 Reflection in Distributed Monitoring

Reflection offers the flexibility necessary to address the distributed monitoring problem. Besides being dynamic enough to allow growing knowledge vectors, sufficient reflection can also allow in-place modification and removal of monitors.

Suppose a distributed system uses the actor framework, monitors could be implemented as meta-actors. Unmonitored actors would remain untouched while actors with installed meta-actors as monitors would have modified semantics. Note that meta-actors, being actors themselves, would use normal actor semantics unless they have installed monitors, and thus meta-actors, of their own.

Using the semantics given in Section 5.2, the meta-actor `transmit` semantics can be overridden to modify the outgoing message to include the knowledge vector, in the case of our monitors.

$$\begin{aligned} \mathbf{transmit} \text{ (on monitor): } & [R[\mathbf{transmit}(t, m)], b]_{\hat{a}} \rightarrow [R[\mathbf{send}(t, m'')], b]_{\hat{a}} \\ & m'' = KVm(m, kv(a)) \end{aligned}$$

In order to deal with the knowledge vectors on a monitor, we simply remove $kv(a)$ and propagate the original message to the recipient actor:

$$\begin{aligned} \mathbf{dlv} \text{ (on monitor): } & [R[\mathbf{dlv}(KVm(m, kv(t))), b]_{\hat{a}} \rightarrow [R[\mathbf{nil}], b]_{\hat{a}}, < a \Leftarrow m >_{\text{meta}} \\ & \text{where } \mathbf{meta}(a) = \hat{a} \end{aligned}$$

Monitors as meta-actors would have access to actors' internal states, also needed for the knowledge vectors. With all of this, the monitors would be able to track distributed system computation and check for errors. Because the meta-actors can reflectively modify themselves during runtime, monitors can be modified or removed on the fly without the need to redeploy monitored processes.

7 Implementation of Adaptive Monitors

A proof-of-concept implementation was done building upon Actor Architecture. Although the implementation language does not provide strong reflective capabilities, reflection using a library and indirection enabled us to achieve our goal of a dynamic distributed monitoring system.

7.1 Actor Architecture

The *Actor Architecture* (AA) is a Java actor framework providing a full actor implementation running on one or more systems [26]. AA handles message routing and actor migration and consists of two main parts: the platform, which provides all of the 'background' services, and the actor itself.

Actors in AA are simply Java subclasses of the `Actor` class. The `ActorThread` sleeps until a message is inserted into its mail queue by the local `MessageManager`. The `ActorThread` then processes the message by parsing the requested method and the corresponding arguments. If the method exists with the correct number of arguments in the actor object, the method is called.

7.2 Monitor Installation

Once an actor is created, a monitor is installed by sending a message. The message simply provides the monitor name and calls the actor class' `addMonitor()` method. This method uses Java's reflective facilities to create a new instance of the class named by the string argument. The newly created monitor object is then added to the list of this actor's monitors.

Although this installation method requires the monitor bytecode to be present on the system, there is no reason this is necessary. It is feasible to serialize the monitor and send it with the message to each actor. However, both schemes are equally flexible and only differ in when the monitor code is transferred. The class also has a method `removeMonitor()` to remove currently installed monitors.

7.3 Knowledge Vectors

The `KnowledgeVector` class is a collection of `KVEntry` objects. Each `KVEntry` contains only a value and a timestamp and the `KnowledgeVector` keeps a mapping between the entries and the associated actor names.

Scattered throughout the `Actor` and `ActorThread` code are calls to the `updateKV(KnowledgeVector)` method. This method call is necessary every time the local knowledge vector can change. Thus, every time a message is received at

an actor, the local knowledge vector is compared with the message's piggybacked knowledge vector and necessary updates are incorporated.

After message processing and the corresponding call have completed, the actor's `updateLocalKV()` method is called in order to assure that the local knowledge vector contains the most current entries for local variables. Ideally, the knowledge vector update would be triggered by modification of any monitored variables but since actors are purely reactive, any modification must occur as a result of a message. Thus, updating the knowledge vector after completion of message processing guarantees that all variable modifications are recorded.

In order to update the local portions of the knowledge vector, the method iterates through each locally monitored variable and reflectively reads it. Java reflection is necessary in order to obtain references to the variables by only the string identifier. The `updateLocalKV()` method then records the current values and the current timestamp in the knowledge vector.

7.4 Actor Monitor

The `ActorMonitor` is an abstract class containing methods and variables related to the monitors. A monitor is created by extending the `ActorMonitor` class and implementing the `evaluateMonitor()` method.

Aside from the evaluate method, there are also two methods for getting and setting the list of monitored variables. Monitored variables are stored as a string that is the concatenation of the actor name or `local` and the variable name. The `local` keyword specifies that the variable will be evaluated at each actor locally whereas the standard actor names specify that the variable will only be evaluated at the specified actor.

```
public MyMonitor(){
    setMonitoredVariable("uan://127.0.0.1:2/counter");
    setMonitoredVariable("local/counter");
}
```

Fig. 5. The monitor's constructor showing manual listing of monitored variables

The `setMonitoredVariable()` modifier is only used within the constructor of the monitor to initially create the list of variables used in the monitor formula. The `getMonitoredVariables()` accessor is used in the above mentioned `updateLocalKV()` method to access the locally monitored variables.

Each time the knowledge vector is updated, the actor evaluates its installed monitors. The list of monitors is iterated through and each one is evaluated in turn using the newly updated values.

The `evaluateMonitor()` method does the actual evaluation of the monitor formula. The method pulls out the most currently known variable values from the local knowledge vector. Using these values, the formula is evaluated and a boolean value stating the result of the formula is returned. In the case where the knowledge vector is too sparse to evaluate the monitor, the method simply gives

up and returns success. In addition to the knowledge vector, the local actor name is also passed in for determining which entries correspond to the local actor.

```
public boolean evaluateMonitor(KnowledgeVector kv, String locActorName){
    try{
        int myCnt = kv.get(localActorName + "/counter").getValue();
        int remoteCnt = kv.get("uan://127.0.0.1:2/m_iSum").getValue();

        if(myCnt > remoteCnt)
            System.err.println("MONITOR FAILS!!!! at " + localActorName);
        else if(myCnt > THRESHOLD)
            aboveTH = true;
        else
            aboveTH = false;
    }
    catch(NullPointerException e){
        // just ignore it if the knowledge vector is not full enough
    }
    return(new KVEntry(timestamp++, new Boolean(aboveTH)));
}
```

Fig. 6. The monitor's evaluate method. This monitor ensures that the counter value at every node is never greater than the counter value at actor 127.0.0.1:2.

The `evaluateMonitor()` method returns a `KVEntry`, which is then added to the node's knowledge vector. This provides means of communication between monitors allowing them to cooperate and communicate state information. In this way, monitors can cooperatively execute model-based monitoring.

7.5 Implementation Discussion

Currently, the `evaluateMonitor()` method and the monitored variables must be written by hand. However, we intend to incorporate automatic generation of these from a provided PT-DTL formula.

This implementation provides dynamic, adaptive distributed monitoring but is still constrained by Java's limited reflection. Objects can be created and passed around and their methods may be called but a more reflective language would allow even greater flexibility. For instance, implementation in Smalltalk would allow program modification in addition to program introspection.

With the Java implementation, any monitor adaptability has to be coded into the monitor. Monitors cannot simply be installed on any distributed application, the application must be designed with monitoring in mind. An analogy would be having to design an application for debugging versus being able to use a symbolic debugger on any application. Because of Java's weak reflective capabilities, all actors in our implementation must contain code to process knowledge vectors whether a monitor is installed or not.

With a sufficient level of reflection, any application can be monitored without any special design through reflectively overriding the appropriate methods. The send and receive methods would not need to handle knowledge vectors in the case where no monitor is installed but would be modified upon monitor installation. A Smalltalk implementation is currently underway that uses reflection and meta-actors. Using these, monitors may be installed on any actor implementing these two methods without any special design to accommodate monitors.

8 Related Work

We have used distributed monitoring as a case study to demonstrate the need for and different faces of reflection. As with any problem, there are many possible solutions, each with its own advantages and disadvantages. Many previous systems have offered various solutions to distributed monitoring utilizing different levels of reflection, allowing varying capabilities.

Chen and Rosu introduce *Monitor Oriented Programming* (MOP), which separates monitor specification from the program, much like AOP [12]. In addition to the program implementation, a formal specification is provided. This specification is translated into runtime monitors that are installed similarly to aspects. Should any constraints be violated, user specified code will run to recover from or report the error. However, also like AOP, only limited reflective capabilities are utilized. Indeed, the current implementation of MOP, JavaMOP, compiles the specifications into AspectJ code and thus is constrained by its limitations.

Another area in which reflection is frequently used is active monitoring. The monitors described thus far in this paper are all passive in that they in no way affect the regular operation of the system but monitor quietly in the background. Active monitors visibly affect and participate in the operation of their monitored nodes. *Synchronization constraints* are one form of active monitor used for distributed synchronization [21]. Meta-actors simply delay delivery of messages until a specified condition is met and then delivery continues as normal.

Similarly Aksit et al. introduce an AOP model permitting composition of multiple filters [2]. *Filters* are meta-level objects that intercept messages to program objects, apply specified conditions and then allow or disallow message delivery or perform a specified action. Program objects can be reified by a filter permitting observation or modification of the object. Filters can be layered to achieve complex functionality while still maintaining logical encapsulation and code reuse much like layering of meta-actors.

Broadway is an extension to C++ simplifying distributed systems development by providing an actor and meta-actor framework [36]. Since C++ is not a reflective language, reflection is accomplished through use of a library and changing function pointers. Using this scheme, a fully reflective actor system is created enabling redefinition of send, receive, and become methods.

Using *Broadway*, Sturman created *DIL*, a high-level language that can be used to define protocols as wrappers to actors [37]. These protocol wrappers are applied and enforced through meta-actors to capture and modify incoming and outgoing messages. Similarly, Astley introduced *Distributed Connection*

Language as a method of connecting distributed components through use of meta-actors to transform and route data [4].

9 Discussion

Reflective programming is, although a powerful tool, difficult to use and complex to reason and verify. Part of the reason reflective facilities are so limited in real languages is this complexity, for both the programmer and the runtime developer. Between this complexity and performance issues, reflection has had limited reach.

As mentioned in Section 3.2, most reflective languages limit the amount of reflection offered. Reflective facilities provide great flexibility but at a cost to performance, security, and ease of programming. Because of the necessity for program and metadata to be mutable, reflective runtime systems must be either interpreted or provide much indirection, both of which reduce performance and efficiency [17].

The flexibility and dynamicity provided by reflection also raise security and correctness concerns. Reflection and metaprogramming have been used to provide security and for verification purposes [7][18][2]. However, these start with the assumption that the reflective system itself is secure and correct. Much work remains to be done on verification and security of reflective systems themselves. Some security work has been done on Java's reflective system [11]. However, different security issues are raised by fully reflective systems, given their dynamicity and the possibility of interference between concurrent applications. Correctness and verification of reflective programs are especially important given the difficulty of programming these dynamic systems.

In general, fully reflective programming is useful for research and experimentation to determine what is useful and turn that into simpler programming tools and paradigms (e.g., AOP, TLAM). We believe that such simpler programming paradigms and tools will be essential to address the complexity of coordination in large-scale concurrent and distributed systems. Such systems are of increasing performance as web services and applications, sensor networks, and multicore architectures continue to grow in importance.

Acknowledgements

We would to thank Carolyn Talcott, MyungJoo Ham, Rajesh Kumar, and Sameer Sundresh for their helpful comments on an earlier draft.

References

1. Agha, G.: *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA (1986)
2. Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A.: *Abstracting Object Interactions Using Composition Filters*. In: Nierstrasz, O. (ed.) *ECOOP 1993*. LNCS, vol. 707, pp. 152–184. Springer, Heidelberg (1993)

3. Arbab, F.: Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science* 14(03), 329–366 (2004)
4. Astley, M. *Customization and Composition of Distributed Objects: Policy Management in Distributed Software Architectures*. PhD thesis, University of Illinois at Urbana-Champaign (1999)
5. Astley, M., Agha, G.: Customization and composition of distributed objects: Middleware abstractions for policy management. In: *Sixth International Symposium on the Foundations of Software Engineering*, ACM SIGSOFT (1998)
6. Astley, M., Sturman, D., Agha, G.: Customizable middleware for modular distributed software. *Communications of the ACM* 44, 99–107 (2001)
7. Bandinelli, S., Fuggetta, A.: Computational reflection in software process modeling: The SLANG approach. In: *Proceedings of 15th International Conference on Software Engineering*, pp. 144–154 (1993)
8. Bawden, A.: Reification without evaluation. In: *LFP '88: Proceedings of the 1988, ACM conference on LISP and functional programming*, pp. 342–349 (1988)
9. Brant, J., Foote, B., Johnson, R.E., Roberts, D.: Wrappers to the Rescue. In: Jul, E. (ed.) *ECOOP 1998*. LNCS, vol. 1445, pp. 396–417. Springer, Heidelberg (1998)
10. Cameron, R., Ito, M.: Grammar-Based Definition of Metaprogramming Systems. *ACM Transactions on Programming Languages and Systems* 6, 1 (1984)
11. Caromel, D., Vayssiere, J.: Reflections on MOPs, Components, and Java Security. In: Knudsen, J.L. (ed.) *ECOOP 2001*. LNCS, vol. 2072, pp. 256–274. Springer, Heidelberg (2001)
12. Chen, F., Rosu, G.: Java-MOP: A monitoring oriented programming environment for Java. In: *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2005)
13. Cointe, P., Amiot, A., Denier, S.: From (meta) objects to aspects: from Java to AspectJ. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) *FMCO 2004*. LNCS, vol. 3657, pp. 70–94. Springer, Heidelberg (2005)
14. Czarnecki, K., Eisenecker, U.W.: *Generative programming*. Springer, Heidelberg (2000)
15. Czarnecki, K., Eisenecker, U.W.: Components and generative programming. In: *Proceedings of 7th European software engineering conference and 7th ACM SIGSOFT symposium on Foundations of software engineering*, pp. 2–19 (1999)
16. Denker, G., Meseguer, J., Talcott, C.: Rewriting semantics of meta-objects and composable distributed services. *Futatsugi* [139], 407–427 (1999)
17. Deutsch, L., Schiffman, A.: Efficient implementation of the smalltalk-80 system. In: *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 297–302 (1984)
18. Fabre, J.C., Perennou, T.: A metaobject architecture for fault-tolerant distributed systems: the FRIENDS approach. *IEEE Transactions on Computers* 47(1), 78–95 (1998)
19. Foote, B., Johnson, R.E.: Reflective facilities in Smalltalk-80. *ACM SIGPLAN Notices* 24(10), 327–335 (1989)
20. Friedman, D., Wand, M.: Reification: Reflection without metaphysics. In: *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (1984)
21. Frølund, S.: Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages. In: Madsen, O.L. (ed.) *ECOOP 1992*. LNCS, vol. 615, pp. 185–196. Springer, Heidelberg (1992)
22. Frølund, S., Agha, G.: A language framework for multi-object coordination. In: Nierstrasz, O. (ed.) *ECOOP 1993*. LNCS, vol. 707, pp. 346–360. Springer, Heidelberg (1993)

23. Green, D.: Trail: The Reflection API. In: *The Java Tutorial Continued: The Rest of the JDK (TM)*. Addison-Wesley Pub. Co., Reading (1998)
24. Hennessy, J.: Symbolic Debugging of Optimized Code. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4(3), 323–344 (1982)
25. Hutto, P., Ahamad, M.: Slow memory: weakening consistency to enhance concurrency in distributed shared memories. In: *Proceedings of 10th International Conference on Distributed Computing Systems*, pp. 302–309 (1990)
26. Jang, M.: *The Actor Architecture Manual* (2004)
27. Kim, W., Agha, G.: Compilation of a highly parallel actor-based language. In: *The Fifth International Workshop on Languages and Compilers for Parallel Computing*, pp. 1–12 (1992)
28. Lea, R., Yokote, Y., Itoh, J.-I.: Adaptive operating system design using reflection. In: *HOTOS '95: Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, p. 95 (1995)
29. Maes, P.: *Computational Reflection*. Springer, London (1987)
30. Mason, I.A., Talcott, C.: A semantically sound actor translation. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) *ICALP 1997*. LNCS, vol. 1256, Springer, Heidelberg (1997)
31. Meseguer, J., Talcott, C.L.: Semantic Models for Distributed Object Reflection. In: Magnusson, B. (ed.) *ECOOP 2002*. LNCS, vol. 2374, pp. 1–36. Springer, Heidelberg (2002)
32. Rivard, F.: Smalltalk: a Reflective Language. In: *Proceedings of Reflection*, pp. 21–38 (1996)
33. Sen, K., Rosu, G., Agha, G.: Runtime safety analysis of multithreaded programs. In: *Proceedings of the 9th European software engineering and 11th ACM SIGSOFT symposium on Foundations of software engineering*, pp. 337–346 (2003)
34. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient Decentralized Monitoring of Safety in Distributed Systems. In: *Proceedings of the 26th International Conference on Software Engineering*, pp. 418–427 (2004)
35. Smith, B.C.: *Reflection and semantics in LISP*. ACM Press, New York (1984)
36. Sturman, D.: Fault-adaptation for systems in unpredictable environments. Master's thesis, University of Illinois at Urbana-Champaign (1994)
37. Sturman, D.: Modular Specification of Interaction Policies in Distributed Computing. PhD thesis, University of Illinois at Urbana-Champaign (1996)
38. Sturman, D., Agha, G.: A protocol description language for customizing failure semantics. In: *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pp. 148–157 (1994)
39. Talcott, C., Venkatasubramanian, N.: A Semantic Framework for Specifying and Reasoning about Composable Distributed Middleware Services (2001)
40. Venkatasubramanian, N., Talcott, C.: Reasoning about meta level activities in open distributed systems. In: *Proceedings of the 14th annual ACM symposium on Principles of distributed computing*, pp. 144–152 (1995)
41. Venkatasubramanian, N., Talcott, C., Agha, G.: A formal model for reasoning about adaptive QoS-enabled middleware. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 13(1), 86–147 (2004)
42. Watanabe, T., Yonezawa, A.: An Actor-Based Metalevel Architecture for Group-Wide Reflection. In: *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pp. 405–425 (1990)

Links: Web Programming Without Tiers

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop*

University of Edinburgh

Abstract. Links is a programming language for web applications that generates code for all three tiers of a web application from a single source, compiling into JavaScript to run on the client and into SQL to run on the database. Links supports rich clients running in what has been dubbed ‘*Ajax*’ style, and supports concurrent processes with statically-typed message passing. Links is *scalable* in the sense that session state is preserved in the client rather than the server, in contrast to other approaches such as Java Servlets or PLT Scheme. Client-side concurrency in JavaScript and transfer of computation between client and server are both supported by translation into continuation-passing style.

1 Introduction

A typical web system is organized in three tiers, each running on a separate computer (see Figure 1). Logic on the middle-tier server generates pages to send to a front-end browser and queries to send to a back-end database. The programmer must master a myriad of languages: the logic is written in a mixture of Java, Perl, PHP, and Python; the pages described in HTML, XML, and JavaScript; and the queries are written in SQL or XQuery. There is no easy way to ensure that data interfaces between the languages match up — that a form in HTML or a query in SQL produces data of a type that the logic in Java expects. This is called the *impedance mismatch* problem. The problem is exacerbated because code for the browser or database is often generated at runtime, making web applications error prone and difficult to debug.

Links eliminates impedance mismatch by providing a single language for all three tiers. In the current version, Links translates into JavaScript to run on the browser and SQL to run on the database. The server component is written in O’Caml; it consists of a static analysis phase (including Hindley-Milner typechecking), a translator to JavaScript and SQL, and an interpreter for the Links code that remains on the server. All run-time code generation is performed by the compiler in a principled manner, rather than by the programmer using techniques such as string processing; principled generation of this code makes applications less error prone and easier to debug, as well as making it now possible to perform type checking for information passed between the three tiers. The programmer still needs to think about which code runs in which location, especially regarding security of data, but now in a context where the fundamental properties of well-formed code and well-typed data are guaranteed.

Increasingly, web applications designers are migrating work into the browser. “Rich client” systems, such as Google Mail and Google Maps, use a new style of interaction

* This research was supported by EPSRC grant number EP/D046769/1.

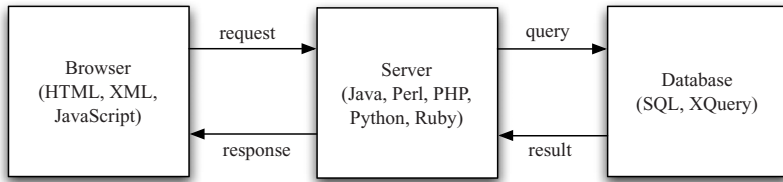


Fig. 1. Three-tier model

dubbed “*Ajax*” [11]. Client-side Links compiles into JavaScript, a functional language widely available on most browsers. JavaScript is notoriously variable across platforms, so we have designed the compiler to target a common subset that is widely available. It would be easy to extend Links to support additional target languages, such as Flash or Java. However, we expect the popularity of Ajax will mean that standard and reliable versions of JavaScript will become available over the next few years.

Links is a strict, typed, functional language. It incorporates ideas proven in other functional languages, including:

- database query optimization, as found in Kleisli and elsewhere,
- continuations for web interaction, as found in PLT Scheme and elsewhere, and
- concurrency with message passing, as found in Erlang and elsewhere,

All three of these features work better with immutable values rather than mutable objects. In Links, side effects play a limited (though important) role, being used for updates to the database and display, and communication between concurrent processes. Types ensure consistency between forms in the browser, logic in the server, and queries on the database—and between senders and receivers in a concurrent program.

Links programs are *scalable* in the sense that session state is preserved in the client rather than the server. Many commercial web tools (like J2EE) and most research web tools (including current releases of PLT Scheme [15] and Mozart QHTML [9]) are not scalable in this sense.

In Links, all server state is serialized and passed to the client, then restored to the server when required. This *resumption passing style* extends the *continuation passing style* commonly used in PLT Scheme and elsewhere. Links functions are labelled as to whether they are intended to execute on the client or the server; functions running on the client may invoke those on the server, and vice-versa.

Database programming. Queries are written in the Links notation and compiled into SQL, a technique pioneered by Kleisli [7][36] and now used in LINQ [18].

Web interaction. The notion that a programming language could provide support for web interaction first appears in the programming language MAWL [1]. The notion of *continuation* from functional programming has been particularly fruitful, being applied by a number of researchers to improve interaction with a web client, including Quiennec [25], Graham [13] (in a commercial system sold to Yahoo and widely used for building web stores), Felleisen and others [14][15], and Thiemann [31].

Concurrency. Links supports concurrent programming in the client, using “share nothing” concurrency where the only way processes can exchange data is by message passing, as pioneered in Erlang [2] and Mozart [33].

XML programming. Links provides convenient syntax for constructing XML data, similar to that provided by XQuery [37]. The current version does not support regular expression types, as found in XDuce and other languages, but we may add them in a future version. Regular expression types were given a low priority, because they are already well understood from previous research [19].

Other languages. Other languages for web programming include Xtatic [16], Scala [22], Mozart [9], SML.NET [5], F# [30], C ω (based on Polyphonic C# [4] and Xen [6]), HOP [29] and Ocsigen [3]. These languages have many overlaps with Links, as they are also inspired by the functional programming community.

However, none of these languages shares Links’ objective of generating code for all three tiers of a web application from a single source — scripts for the front-end client, logic for the middle-tier server, and queries for the back-end database. We expect that providing a single, unified language as an alternative to the current multiplicity of languages will be a principal attraction of Links.

This paper. We introduce Links by describing three examples in Section 2. Section 3 sketches our features for concurrency and client-server interaction. Section 4 gives an SQL-compilable subset of Links and details how it is compiled into SQL, while Section 5 describes “mailbox typing” for message-passing. Section 6 discusses some shortcomings of the current implementation and Section 7 concludes.

2 Links by Example

This section introduces Links by a series of examples. The reader is encouraged to try these examples online at

`http://groups.inf.ed.ac.uk/links/examples/`

We begin with an example to introduce the basic functionality of Links, and then present two further examples that demonstrate additional capabilities: a draggable list, and a progress bar.

2.1 Dictionary Suggest

The Dictionary Suggest application presents a text box, into which the user may type a word. As the user types, the application displays a list of words that could complete the one being typed. A dictionary database is searched, and the first ten words beginning with the given prefix are presented (see Figure 2). In addition, the user can add, update and delete a definition by clicking on it. To add a new definition the user fills in and submits the form at the bottom of the page by clicking ‘Add’.

Dictionary suggest

Search for definitions

Click a definition to edit it

[fun](#) Sport, merriment, frolicsome amusement.
[funambulate](#) To walk or to dance on a rope.
[funambulation](#) Ropedancing.
[funambulatory](#) Narrow, like the walk of a ropedancer.
[funambulatory](#) Performing like a ropedancer.
[funambulist](#) A ropewalker or ropedancer.
[funambulo](#) Alt. of Funambulus
[funambulus](#) A ropewalker or ropedancer.

Word:

Meaning:

[function](#) A quantity so connected with another quantity, that if any alteration be made in the latter there will be a consequent alteration in the former. Each quantity is said to be a function of the other. Thus, the circumference of a circle is a function of the diameter. If x be a symbol to which different numerical values can be assigned, such expressions as x^2 , $3x$, $\text{Log } x$, and $\text{Sin } x$, are all functions of x .

New definition

Word:

Meaning:

Fig. 2. Dictionary Suggest screenshot. An entry for “function” appearing in the word list has just been clicked, so its entry has expanded into an editable form.

To update an existing definition the user clicks on one of the suggestions, which then expands into a form containing the existing definition; then the user edits the form and submits it by clicking ‘Update’. This form also includes buttons for cancelling the update (which restores the original suggestion) and deleting the definition.

This application is of interest because it must perform a database lookup and update the display at every keystroke. Applications such as Google Suggest [17] have a similar structure.

The Links version is based on an ASP.NET version, available online [21], using the same data. It extends the ASP.NET version by allowing the definitions to be added, updated and deleted. The dictionary contains 99,320 entries. For comparison, the ASP.NET version responds to a keystroke in about 0.1s (due to the difficulty of instrumenting third-party code, measurements were made with a stopwatch. The average of a dozen measurements, allowing for a similarly measured reaction time, was 0.1s). In the Links version, over 36 trials with various prefixes, total response time measured on average 649ms with a standard deviation of 199ms; subtracting the time spent performing the database query in each trial, the average time taken was 297ms with a standard

```

var defsTable =
  table "definitions" with
    (id:String, word:String, meaning:String)
  where id readonly from database "dictionary";

fun newDef(def) server { insert defsTable values [def] }
fun updateDef(def) server {
  update (var d <-- defsTable) where (d.id == def.id)
  set (word=def.word, meaning=def.meaning)
}
fun deleteDef(id) server {
  delete (var def <-- defsTable) where (def.id == id)
}

fun completions(s) server {
  if (s == "") [] else {
    take(10, for (var def <-- defsTable)
      where (def.word ~ /s.*/)) orderby (def.word)
    [def])
  }
}

fun suggest(s) client {
  replaceChildren(format(completions(s)),
    getNodeById("suggestions"))
}

fun editDef(def) client {
  redraw(
    <form l:onsubmit="{
      var def = (id=def.id, word=w, meaning=m); updateDef(def);
      redraw(formatDef(def), def.id)}" method="POST">
    <table>
      <tr><td>Word:</td><td>
        <input l:name="w" value="{def.word}"/></td></tr>
      <tr><td>Meaning:</td><td>
        <textarea l:name="m" rows="5" cols="80">
          {stringToXml(def.meaning)}</textarea></td></tr>
    </table>
    <button type="submit">Update</button>
    <button l:onclick="{redraw(formatDef(def), def.id)}">
      Cancel</button>
    <button l:onclick="{deleteDef(def.id); redraw([], def.id)}">
      style="position:absolute; right:0px">Delete</button>
    </form>,
    def.id)
}

```

Fig.3. Dictionary suggest in Links (1)

```

fun redraw(xml, defId) client {
  replaceChildren(xml, getNodeById("def:" ++ defId))
}

fun formatDef(def) client {
  <span l:onclick="{editDef(def)}">
    <b>{stringToXml(def.word)}</b>
    {stringToXml(def.meaning)}<br/>
  </span>
}

fun format(defs) client {
  <#>
  <h3>Click a definition to edit it</h3>
  for (var def <- defs)
    <span class="def" id="def:def.id">{formatDef(def)}</span>
  </#>
}

fun addForm(handler) client {
  <form l:onsubmit="{handler!NewDef((word=w, meaning=m))}">
    <table>
      <tr><td>Word:</td><td>
        <input type="text" l:name="w"/></td></tr>
      <tr><td>Meaning:</td><td>
        <textarea l:name="m" rows="5" cols="80"/></td></tr>
      <tr><td><button type="submit">Add</button></td></tr>
    </table>
  </form>
}

var handler = spawn {
  fun receiver(s) {
    receive {
      case Suggest(s) -> suggest(s); receiver(s)
      case NewDef(def) ->
        newDef(def);
        replaceChildren(addForm(self()), getNodeById("add"));
        suggest(s); receiver(s)
    }
  }
  receiver("")
};

```

Fig. 4. Dictionary suggest in Links (2)

deviation of 54ms. Given that no effort has been spent trying to optimize the Links system, this seems to indicate acceptable performance at this stage.

```

<html>
  <head>
    <style>.def {{ color:blue }}</style>
    <title>Dictionary suggest</title>
  </head>
  <body>
    <h1>Dictionary suggest</h1>
    <h3>Search for definitions</h3>
    <form l:onkeyup="{handler!Suggest(s)}">
      <input type="text" l:name="s" autocomplete="off"/>
    </form>
    <div id="suggestions"/>
    <h3>New definition</h3>
    <div id="add">{addForm(handler)}</div>
  </body>
</html>

```

Fig. 5. Dictionary suggest in Links (3)

The code for the application is shown in Figures 3-5; following is a short walk-through of the code. On each keystroke, a *Suggest* message containing the current contents of the text field is sent to the *handler* process. The *handler* process passes the text content to the function *suggest*. This function calls *completions*, which executes on the server, to find the first ten words with the given prefix, and *format* (executing on the client) to format the list returned. Doing the server interaction in a separate *handler* process allows the user interaction to remain responsive, even while looking up suggestions.

The rest of the code is concerned with modifying the database. A form for adding definitions is created by the function *addForm*. Clicking ‘Add’ sends a *NewDef* message to the *handler* process containing a new definition. The *handler* process calls *newDef* to add the definition, then resets the form and updates the list of suggestions (in case the new definition appears in the current list of suggestions).

Clicking on a definition invokes the function *editDef*, which calls the function *redraw* in order to replace the definition with a form for editing it. Clicking ‘Cancel’ reverses this operation. Clicking ‘Update’ or ‘Delete’ performs the corresponding modification to the definition by calling *updateDef* or *deleteDef* on the server, and then updates the list of suggestions by calling the function *redraw* on the client.

Having sketched the basic structure of the example, we now describe several of the key features of Links, illustrating them by our example code. The features we cover are syntax, types, XML, regular expressions, interaction, list comprehensions, database access and update, concurrency, and partitioning the program onto client and server.

The core of Links is a fairly standard functional programming language with Hindley-Milner type inference. One missing feature is exception handling, which we plan to add in a future version.

Syntax. The syntax of Links resembles that of JavaScript. This decision was made not because we are fond of this syntax, but because we believe it will be familiar to our

target audience. Low-level syntactic details become a matter of habit that can be hard to change: we conjecture that using the familiar $f(x, y)$ in place of the unfamiliar $f\ x\ y$ will significantly lower the barrier to entry of functional programming.

One difference from JavaScript syntax is that we do not use the keyword **return**, which is too heavy to support a functional style. Instead, we indicate return values subtly (perhaps too subtly), by omitting the trailing semicolon. The type checker indicates an error if a semicolon appears after an expression that returns any value other than the unit value $()$.

Types. Links uses Hindley-Milner type inference with row variables [24]. As basic types Links supports integers, floats, characters, booleans, lists, functions, records, and variants.

- A list is written $[e_1, \dots, e_k]$, and a list type is written $[A]$.
- A lambda abstraction is written **fun** $(x_1, \dots, x_k) \{e\}$, and a function type is written $(A_1, \dots, A_k) \rightarrow B$.
- A record is written $(f_1=e_1, \dots, f_k=e_k)$, and a record type is written $(f_1:A_1, \dots, f_k:A_k \mid r)$, where r is an optional row variable. Field names for records begin with a lower-case letter.
- A variant is written $F_i(e_i)$ and a variant type is written $[F_1:A_1, \dots, F_k:A_k \mid r]$. Field names for variants begin with an upper-case letter.

Strings are simply lists of characters, and tuples are records with natural number labels. Apart from the table declaration in Figure 3 none of the examples in the paper explicitly mention types. This is partly because type inference renders type annotations unnecessary and partly to save space. Nevertheless, it should be stressed that all of the examples are type-checked statically, and static typing is an essential part of Links.

Links currently does not support any form of overloading; we expect to support overloading in future using a simplified form of type classes. As with regular expression types, this was left to the future because it is well understood from other research efforts. In particular, the WASH and iData systems make effective use of type classes to support generic libraries [31][23].

XML. Links includes special syntax for constructing and manipulating XML data. XML data is written in ordinary XML notation, using curly braces to indicate embedded code. Embedded code may occur either in attributes or in the body of an element. The Links notation is similar to that used in XQuery, and has similar advantages. In particular, it is easy to paste XML boilerplate into Links code. The parser begins parsing XML when a $<$ is immediately followed by a legal tag name; a space must always follow $<$ when it is used as a comparison operator; legal XML does not permit a space after the $<$ that opens a tag. Links also supports syntactic sugar $<\#\dots\#>$ for specifying an XML forest literal as in the function *format* in Figure 4.

The client maintains a data structure representing the current document to display, called the Document Object Model, or DOM for short. Often this structure is represented in some form of HTML, such as XHTML, the dialect of XML corresponding to HTML. Links provides library functions to access and modify the DOM, based on the

similar operations specified by the W3C DOM standard. Links supports two types for manipulating XML: *DomNode* is a mutable reference (similar to a *ref* type in ML), while *Xml* is an immutable list of trees. There is an operation that converts the former to the latter by making a deep copy of the tree rooted at the node, returning it in a singleton list. We expect eventually to support regular expression types for XML that refine each of these two types, and to support a notation like XPath for manipulating trees, but as these points are well understood (but a lot of work to implement) they are not a current priority.

Regular expressions. Matching a string against a regular expression is written $e \sim /r/$ where r is a regular expression. Curly braces may be used to interpolate a string into a regular expression, so for example $/\{s\} . */$ matches any string that begins with the value bound to the variable s .

Interaction. The Links code specifies display of an XML document, the crucial part of which is the following:

```
<form l:onkeyup="{handler!Suggest(s)}">
  <input type="text" l:name="s" autocomplete="off"/>
</form>
<div id="suggestions"/>
```

The `l:name` attribute specifies that the string typed into the field should be bound to a string variable s .

The attributes `l:name` and `l:onkeyup` are special. The attribute `l:onkeyup` is followed by Links code in curly braces that is *not* immediately evaluated, but is evaluated whenever a key is released while typing in the form. (Normally, including curly braces in XML attributes, as elsewhere in XML, causes the Links code inside the braces to be evaluated and its value to be spliced into the XML.) The `l:name` attribute on an input field must contain a Links variable name, and that variable is bound to the contents of the input field.

The attributes that Links treats specially are `l:name` and all the attributes connected with events: `l:onChange`, `l:onsubmit`, `l:onkeyup`, `l:onmousemove`, and so on. These attributes are prefixed with `l:`, using the usual XML notation for namespaces; in effect, `l` denotes a special Links namespace.

The scope of variables bound by `l:name` is the Links code that appears in the attributes connected with events. Links static checking ensures that a static error is raised if a name is used outside of its scope; this guarantees that the names mentioned on the form connect properly to the names referred to by the application logic. In this, Links resembles MAWL and Jwig, and differs from PLT Scheme or PHP.

Our experience has shown that this style of interaction does not necessarily scale well, and it may be preferable to use a library of higher-order forms as developed in WASH or iData. We return to this point in Section [6](#).

List comprehensions. The Links code calls the function *suggest* each time a key is released. This in turn calls *completions* to find the first ten completions of the prefix, and *format* to format the results as HTML for display. Both *completions*

and *format* use **for** loops, in the former case also involving **where** and **orderby** clauses. These constructs correspond to what is written as a *list comprehension* in languages such as Haskell or Python. Each comprehension is equivalent to an ordinary expression using standard library functions, we give three examples, where the comprehension is on the left and its translation is on the right.

```

for (var x <- e1)          concatMap(fun(x) {e2}, e1)
  e2

for (var x <- e1)          concatMap(
  where (e2)                  fun(x) {if (e2) e3 else []},
  e3                          e1)

for (var x <- e1)          concatMap(
  orderby (e2)                 fun(x) {e3},
  e3                          orderBy(fun(x) {e2}, e1)

```

Here *concatMap*(*f*, *xs*) applies function *f* to each element in list *xs* and concatenates the results, and *orderBy*(*f*, *xs*) sorts the list *xs* so that it is in ascending order after *f* is applied to each element. The **orderby** notation is conceptually easier for the user (since there is no need to repeat the bound variables) and technically easier for the compiler (because it is closer to the SQL notation that we compile into, as discussed in the next section; indeed, currently **orderby** clauses only work reliably in code that compiles into SQL, because the absence of overloading means we have not yet implemented comparison on arbitrary types in the client or server).

Database. Links provides facilities to query and update SQL databases, where database tables are viewed as lists of records. We hope to provide similar facilities for XQuery databases in the future.

Links provides a **database** expression that denotes a connection to a database, and a **table** expression that denotes a table within a database. A database is specified by name (and optional configuration data, which is usually read from a configuration file), and a table is specified by the table name, the type signature of a row in the table, and the database containing the table.

The type of a table is distinct from the type of its list of records, since tables (unlike lists) may be updated. The coercion operation *asList* takes a table into the corresponding list, and **for** (**var** *x* <-< *e1*) *e2* (with a long arrow) is equivalent to **for** (**var** *x* <- *asList*(*e1*)) *e2* (with an ordinary arrow).

In the following example, there is a table of words, where each row contains three string fields: the word itself, its type (noun, verb, and so on), and its meaning (definition). Typically, one expresses queries using the Links constructs such as **for**, **where**, and **orderby**, and functions on lists such as *take* and *drop*. The Links compiler is designed to convert these into appropriate SQL queries over the relevant tables whenever possible. For example, the expression

```

take(10, for (var def <-< defsTable)
  where (def.word ~ /s.*/) orderby (def.word)
  [def])

```

compiles into the equivalent SQL statement

```

SELECT def.meaning AS meaning, def.word AS word
FROM definitions AS def
WHERE def.word LIKE '{s}%'
ORDER BY def.word ASC
LIMIT 10 OFFSET 0

```

This form of compilation was pioneered in systems such as Kleisli [7,36], and is also central to Microsoft's new Language Integrated Query (LINQ) for .NET [18]. A related approach, based on abstract interpretation rather than program transformation, is described by Wiederman and Cook [35].

Note that the match operator \sim on regular expressions in Links is in this case compiled into the `LIKE` operator of SQL, and that the call to `take` in Links is compiled into `LIMIT` and `OFFSET` clauses in SQL. At run time, the phrase `{s}` in the SQL is replaced by the string contained in the variable `s`, including escaping of special characters where necessary. Links also contains statements to update, delete from, and insert into a database table, closely modeled on the same statements in SQL.

The `LIMIT` and `OFFSET` clauses are not part of the SQL standard, but are available in PostgreSQL, MySQL, and SQLite, the three targets currently supported by Links. For non-standard features such as this, Links can generate different code for different targets; for instance, it generates different variants of `INSERT` for PostgreSQL and MySQL.

Section 4 presents a subset of Links that is guaranteed to compile into SQL queries.

Concurrency. Links allows one to spawn new processes. The expression `spawn {e}` creates a new process and returns a fresh process identifier, the primitive `self()` returns the identifier of the current process, the command `e1 ! e2` sends message `e2` to the process identified by `e1`, and the expression

```

receive {
  case p1 -> e1
  ...
  case pn -> en
}

```

extracts the next message sent to the current process (or waits if there is no message), and executes the first case where the pattern matches the message. (Unlike Erlang, it is an error if no case matches, which fits better with our discipline of static typing.)

By convention, the messages sent to a process belong to a variant type. Event handlers are expected to execute quickly, so we follow a convention that each handler either sends a message or spawns a fresh process. For instance, in the Dictionary Suggest application, each event handler sends a message to a separate handler process that is responsible for finding and displaying the completions of the current prefix. This puts any delay resulting from the database lookup into a separate process, so any keystroke will be echoed immediately. Furthermore, the messages received by the handler process are processed sequentially, ensuring that the updates to the DOM happen in the correct order.

As well as any new processes spawned by the user, there is one distinguished process: the *main* process. The top-level program and all event handlers are run in the main

process. Having all event handlers run in a single process allows us to guarantee that events are processed in the order in which they are received.

Client-server. The keyword **server** in the definition of *completions* causes invocations of that function to be evaluated on the server rather than the client, which is required because the database cannot be accessed directly from the client.

When the Dictionary Suggest application is invoked, the server does nothing except to transmit the Links code (compiled into JavaScript) to the client. If the user presses and releases a key then the *suggest* function will continue to run on the client as its definition is annotated with the keyword **client**. The client runs autonomously until *completions* is called, at which point the *XMLHttpRequest* primitive of JavaScript is used to transmit the arguments to the server and creates a new process on the server to evaluate the call. No server resources are required until *completions* is called. This contrasts with Java Servlets, which keep a process running on the server at all times, or with PLT Scheme, which keeps a continuation cached on the server.

Location annotations **server** and **client** are only allowed on top-level function definitions. If a top-level function definition does not have a location annotation then it will be compiled for both the client and the server. A location annotation should be read as “this function *must* be run in the specified location”. The implementation strategy for client-server communication is discussed further in Section 3.

2.2 Draggable List

The draggable list application demonstrates the use of the concurrency primitives in Links to manage the state of an interactive GUI component. It displays an itemized list on the screen. The user may click on any item in the list and drag it to another location in the list (see Figure 6).

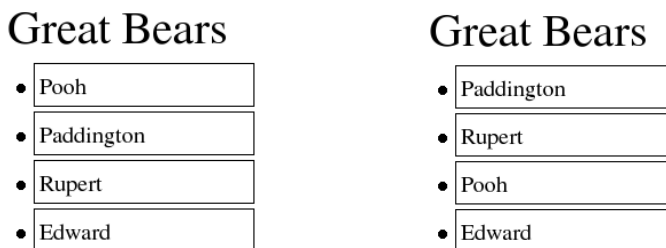


Fig. 6. Draggable list: before and after dragging

The code for the draggable list is shown in Figure 7. The example exploits Links’ ability to run concurrent processes in the client. Each draggable list is monitored by a separate process.

For each significant event on an item in the list (mouse up, mouse down, mouse out) there is a bit of code that sends a message to the right process, indicating the event. The

process itself is coded as two mutually recursive functions, corresponding to two states. The process starts in the waiting state; when the mouse is depressed it changes to the dragging state; and when the mouse is released it reverts to the waiting state. When the mouse is moved over an adjacent item while in the dragging state, the dragged item and the adjacent item are swapped.

Both functions take, as parameter, the DOM `id` of the draggable-list element, thus they know what part of the DOM they control. The `dragging` function takes an additional parameter, designating the particular item that is being dragged. Both functions are written in tail recursive style, each one calling either itself (to remain in that state) or the other (to change state). This style of coding state for a process was taken from Erlang.

In this simple application, the state of the list can be recovered just by examining the text items in the list. In a more sophisticated application, one might wish to add another parameter representing the abstract contents of the list, which might be distinct from the items that appear in the display.

Observe that the code allows multiple lists to coexist independently, each monitored by its own process.

2.3 Progress Bar

The progress bar application demonstrates symmetric client and server calls. (Note that unlike standard Ajax frameworks, in Links client code may call server code and vice-versa, equally easily.) Here some computation is performed on the server, and the progress of this computation is demonstrated with a progress bar (see Figure 8). When the computation is completed, the answer is displayed (see Figure 9).

The code for the progress bar application is shown in Figure 10. In this case, the computation performed on the server is uninteresting, simply counting up to the number typed by the user. In a real application the actual computation chosen might be more interesting.

Periodically, the function `computation` (running on the server) invokes the function `showProgress` (running on the client) to update the display to indicate progress. When this happens, the state of the computation on the server is pickled and transmitted to the client, exploiting continuation-passing style. The client is passed just the name of a function to call, its arguments, and an object representing the server-side continuation to be invoked when the client call is finished. Note that no state whatsoever is maintained on the server when computation is moved to the client. The implementation is discussed in more detail in Section 3.

One advantage of this design is that if the client quits at some point during the computation (either by surfing to another page, terminating the browser, or taking a hammer to the hardware), then no further work will be required of the server.

On the other hand, the middle of an intensive computation may not be the best time to pickle the computation and ship it elsewhere. A more sophisticated design would asynchronously notify the client while continuing to run on the server, terminating the computation if the client does not respond to such pings after a reasonable interval. This is not possible currently, because the design of Links deliberately limits the ways in which the server can communicate with the client, in order to guarantee that long-term

```

fun waiting(id) {
  receive {
    case MouseDown(elem) ->
      if (isElementNode(elem)
          && (parentNode(elem) == getNodeById(id)))
        dragging(id,elem)
      else waiting(id)
    case MouseUp -> waiting(id)
    case MouseOut(newElem) -> waiting(id)
  }
}

fun dragging(id,elem) {
  receive {
    case MouseUp -> waiting(id)
    case MouseDown(elem) ->
      if (isElementNode(elem)
          && (parentNode(elem) == getNodeById(id)))
        dragging(id,elem)
    case MouseOut(toElem) ->
      if (isElementNode(toElem)
          && (parentNode(elem) == getNodeById(id)))
        {swapNodes(elem,toElem); dragging(id,elem)}
      else dragging(id,elem)
  }
}

fun draggableList (id,items) client {
  var dragger = spawn { waiting(id) };
  <ul id="{id}"
    l:onmouseup="{dragger!MouseUp}"
    l:onmouseuppage="{dragger!MouseUp}"
    l:onmousedown="{dragger!MouseDown(getTarget(event))}"
    l:onmouseout="{dragger!MouseOut(getToElement(event))}">
    { for (var item <- items) <li>{ item }</li> }
  </ul>
}

<html><body>
<h1>Draggable lists</h1>
<h2>Great Bears</h2>
{
  draggableList("bears",
                ["Pooh", "Paddington", "Rupert", "Edward"])
}
</body></html>

```

Fig. 7. Draggable lists in Links



Fig. 8. Progress bar

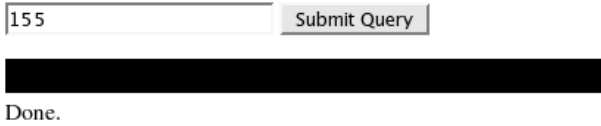


Fig. 9. Progress bar displaying the final answer

session state is maintained on the client rather than the server. This is not appropriate in all circumstances, and future work will need to consider a more general design.

3 Client-Server Computing

A Links program can be seen as a distributed program that executes across two locations: a client (browser) and a server. The programmer can optionally annotate a function definition to indicate where it should run. As mentioned before, code on the client can invoke server code, and vice-versa.

This symmetry is implemented on top of the asymmetric mechanisms offered by web browsers and servers. Current standards only permit the browser to make a direct request to the server. The server can return a value to the browser when done, but there is no provision for the server to invoke a function on the browser directly—so implementing our symmetric calls requires some craft.

Our implementation is also *scalable*, in the sense that session state, when it is captured automatically, is preserved in the client, thus requiring no server resources except when the server is actively working. This is significant since server resources are at a premium in the web environment.

We achieve this by using a variation of continuation-passing style, which we call *resumption-passing style*. It is now common on the web to use continuations to “invert back the inversion of control” [26], permitting a server process to retain control after sending a *form* to the client. But Links permits a server to retain control after invoking an *arbitrary function* on the client.

Figure 11 shows how the call/return style of programming offered by Links differs from the standard request/response style, and how we use request/response style to emulate the call/return style. The left-hand diagram shows a sequence of calls between functions annotated with “client” and “server.” The solid line indicates the active thread of control as it descends into these calls, while the dashed line indicates a stack frame which is waiting for a function to return. In this example, `main` is a client function


```

fun compute(count, total) server {
  if (count < total) {
    showProgress(count, total);
    compute(count+1, total)
  } else "done counting to " ++ intToString(total)
}

fun showProgress(count, total) client {
  var percent =
    100.0 *. intToFloat(count) /. intToFloat(total);
  replaceNode(
    <div id="bar"
      style="width:floatToString(percent)%;
        background-color: black">|</div>,
    getNodeById("bar")
  )
}

fun showAnswer(answer) client {
  domReplaceNode(
    <div id="bar">{stringToXml(answer)}</div>
    getNodeById("bar")
  );
}

<html>
  <body id="body">
    <form l:onsubmit=
      "{showAnswer(compute(0,stringToInt(n)))}">
      <input type="text" l:name="n"/>
      <input type="submit"/>
    </form>
    <div id="bar"/>
  </body>
</html>

```

Fig. 10. Progress bar in Links

which calls a server function f which in turn calls a client function g . The semantics of this call and return pattern are familiar to every programmer.

The right-hand diagram shows the same series of calls as they actually occur at run-time in our implementation. The dashed line here indicates that some local storage is being used as part of this computation. During the time when g has been invoked but has not yet returned a value, the server stores nothing locally, even though the language provides an illusion that f is “still waiting” on the server. All of the server’s state is encapsulated in the value k , which it passed to the client with its call.

To accomplish this, the program is compiled to two targets, one for each location, and begins running at the client. In this compilation step, server-annotated code is replaced on the client by a remote procedure call to the server. This RPC call makes an

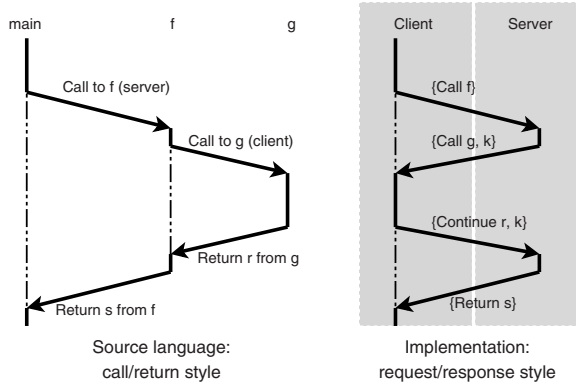


Fig. 11. Semantic behaviour of client/server annotations

HTTP request indicating the server function and its arguments. The client-side thread is effectively suspended while the server function executes.

Likewise, on the server side, a client function is replaced by a stub. This time, however, the stub will not make a direct call to the client, since in general web browsers are not addressable by outside agents. However, since the server is always working on behalf of a client request, we have a channel on which to communicate. So the stub simply gives an HTTP response indicating the server-to-client call, along with a representation of the server’s continuation, to be resumed when the server-to-client call is complete. Upon returning in this way, all of the state of the Links program is present at the client, so the server need not store anything more.

When the client has completed the server-to-client call, it initiates a new request to the server, passing the result and the server continuation. The server resumes its computation by applying the continuation.

3.1 Client-Side Concurrency

A Links program is concurrent: many threads can run simultaneously within the client or within the server, communicating only through message queues.¹ We implement concurrency in the client, even though JavaScript does not offer any concurrency primitives. To accomplish this, we use the following techniques:

- compiling to continuation-passing style,
- inserting explicit context-switch instructions,
- placing any server calls asynchronously (with *XMLHttpRequest*),
- eliminating the stack between context switches using either *setTimeout* or a trampoline.

Producing JavaScript code in CPS allows us to reify each process’s control state, but since JavaScript does not implement tail-call elimination, there is a risk of overflowing the stack.

¹ Note that presently there is no message-based communication between client and server.

To manage the stack and to schedule threads, the compiler wraps every function application and every continuation application in calls to a special “yield” function which does the dirty work. Most calls to `_yield` are simply no-ops, but after every `_yieldGranularity` calls, it will collapse the JavaScript stack and may schedule a new thread to run.

A quick aside about the concurrency semantics: Normally, threads can pre-empt one another at any time. However, inside an event handler, we disallow pre-emption—this allows events to be handled in the order they arrive, for otherwise one event handler might be pre-empted by a later event even before the earlier handler noted the event’s occurrence! In order to prevent event handlers from blocking concurrency, we recommend that they be short-lived (ideally they just send a message or spawn a new process).

With this in mind, we use two techniques for managing the stack, one of which also allows a waiting thread to run; we call these the “timeout technique” and the “exception trampoline.” The latter does not transfer control to other threads and is used during synchronous execution, as with event handlers.

To see these techniques, refer to the implementation of `_yield` for function application² in Figure 12. In detail, the two techniques work as follows:

- In the timeout technique, the thread relinquishes control by returning completely to the top level (that is, to the browser), but only after calling the built-in `setTimeout` function, which adds a given function to a pool of thunks, each of which will be invoked no sooner than the given interval. Since the JavaScript code is all in continuation-passing style, returning from any function returns from the whole call stack, thus emptying it—but the timeout thunk has captured the continuation of that thread, so nothing is lost. The thunk we use simply manages the “process id” global and then calls $f(a, k)$ to continue the thread.

Besides giving other timeout thunks the chance to run, returning to the top level allows any event handlers or any `XMLHttpRequest` callbacks to run.

The `_sched_pause` is a lower bound on how long to wait before allowing this callback to run. Ideally this value should be zero, so there would be no delay between the moment a thread yields and the moment it is eligible to run again. But for some browsers a `_sched_pause` of zero runs the callback immediately, rather than putting it in the pool, thus blocking other threads.

- In the “exception trampoline” method, an exception is thrown containing the current continuation. Throwing the exception unwinds the stack; the trampoline then catches the exception and invokes the continuation. The trampoline code is shown in Figure 13.

Unsurprisingly, preliminary tests indicate that the output of the CPS-translating compiler is slower than comparable direct-style code. The performance penalty appears to be significant but not disastrous: the total run-time of compute-intensive programs we measured was within an order of magnitude of equivalent hand-written JavaScript versions.

² The “yield” function for continuation application is the same except it takes as arguments continuation k and value a , and the function applications $f(a, k)$ are replaced with continuation applications $k(a)$.

```

function _yield(f, a, k) {
  ++_yieldCount;
  if ((_yieldCount % _yieldGranularity) == 0) {
    if (!_handlingEvent) {
      var current_pid = _current_pid;
      setTimeout((function() {
        _current_pid = current_pid;
        f(a, k)}),
        _sched_pause);
    }
    else {
      throw new _Continuation(function () { f(a,k) });
    }
  }
  else {
    return f(a,k);
  }
}

```

Fig. 12. The yield function

Client-to-server calls are implemented using the asynchronous mode of the function *XMLHttpRequest*, to which we pass a callback function that invokes the continuation of the calling process. Using the asynchronous mode allows threads to run via *setTimeout* or triggering user-event handlers.

3.2 Related Work: Continuations for Web Programming

The idea of using continuations as a language-level technique to structure web programs has been discussed in the literature [25][26] and used in several web frameworks (such as Seaside, Borges, PLT Scheme, RIFE and Jetty) and applications, such as ViaWeb's e-commerce application [12], which became Yahoo! Stores. Most these take advantage of a call/cc primitive in the source language, although some implement a continuation-like object in other ways. Each of these systems creates URLs which correspond to continuation objects. The most common technique for establishing this correspondence is to store the continuation in memory, indexed by a unique identifier which is included as part of the URL.

Relying on in-memory tables makes such a system vulnerable to system failures and difficult to distribute across multiple machines. Whether stored in memory or in a database, the continuations can require a lot of storage. Since URLs can live long in bookmarks, emails, and other media, it is impossible to predict how long a continuation should be retained. Most of the above frameworks destroy a continuation after a set period of time. Even with a modest lifetime, the storage cost can be quite high, as each request may generate many continuations and there may be many simultaneous users.

Our approach differs from these implementations by following the approach described by the PLT Scheme team [14]. Rather than storing a continuation object at

```

function _Continuation(v) { this.v = v }

function _wrapEventHandler(handler) {
  return function(event) {
    var active_pid = _current_pid;
    _current_pid = _mainPid;
    _handlingEvent = true;
    var cont = function () { handler(event) }
    for (;;) {
      try {
        cont();
        _handlingEvent = false;
        _current_pid = active_pid;
        return;
      }
      catch (e) {
        if (e instanceof _Continuation) {
          cont = e.v;
          continue;
        }
        else {
          _handlingEvent = false;
          _current_pid = active_pid;
          throw e;
        }
      }
    }
  }
}

```

Fig. 13. The event handler trampoline

the server and referring to it by ID, we serialize continuations (or at least, their “data” portions), embedding them completely in URLs and hidden form variables. The resulting representation includes only the code-pointers and free-variable bindings that are needed in the future of the computation.

We have not yet addressed the issue of security. Security is vital. Our current implementation completely exposes the state of the application to the client. A better system needs to cryptographically encode any sensitive data when it is present on the client.

4 Query Compilation

A subset of Links expressions compiles precisely to a subset of SQL expressions. Figure 14 gives the grammar of the SQL-compilable subset of Links expressions and Figure 15 gives the grammar for the subset of SQL to which we compile.

All terms are identified up to α -conversion (that is, up to renaming of bound variables, field names and table aliases). We allow free variables in queries, though they

are not strictly allowed in SQL: values for these variables will be supplied either at compile time, during query rewriting, or else at runtime. We use vector notation \bar{v} to mean v_1, \dots, v_k , where $1 \leq k$ and abuse vector notation in the obvious way in order to denote tuples, record patterns and lists of tables. For uniformity, we write an empty ‘from’ clause as **from** \bullet . In standard SQL the clause would be omitted altogether. We assume a single database db , and write **table** t **with** (f_1, \dots, f_k) for $asList(\mathbf{table} \ t \ \mathbf{where} \ (f_1:A_1, \dots, f_k:A_k) \ \mathbf{from} \ db)$.

(expressions)	$e ::= take(n, e) \mid drop(n, e) \mid s$
(simple expressions)	$s ::= \mathbf{for} \ (pat \leftarrow s) \ s$ $\quad \mid \ \mathbf{let} \ x = b \ \mathbf{in} \ s$ $\quad \mid \ \mathbf{where} \ (b) \ s$ $\quad \mid \ \mathbf{table} \ t \ \mathbf{with} \ \bar{f}$ $\quad \mid \ [(\bar{b})]$
(basic expressions)	$b ::= b_1 \ op \ b_2$ $\quad \mid \ \mathbf{not} \ b$ $\quad \mid \ x$ $\quad \mid \ lit$ $\quad \mid \ z.f$
(patterns)	$pat ::= z \mid (\bar{f}=\bar{x})$
(operators)	$op ::= \mathbf{like} \mid > \mid = \mid < \mid <> \mid \mathbf{and} \mid \mathbf{or}$
(literal values)	$lit ::= \mathbf{true} \mid \mathbf{false} \mid string\text{-}literal \mid n$
(finite integers)	i, m, n
(field names)	f, g
(variables)	x, y
(record variables)	z
(table names)	t

Fig. 14. Grammar of an SQL-compilable subset of Links

Any expression e can be transformed into an equivalent SQL query by repeated application of the rewrite rules given in Figure 16. To give the rewriting process sufficient freedom, we extend the non-terminal s from Figure 14 to give ourselves a working grammar.

(queries)	$q ::= \text{select } Q \text{ limit } ninf \text{ offset } n$
(query bodies)	$Q ::= cols \text{ from } tables \text{ where } c$
(column lists)	$cols ::= \bar{c}$
(table lists)	$tables ::= \bar{t} \text{ as } \bar{a} \mid \bullet$
(SQL expressions)	$c, d ::= c \text{ op } d$ $\quad \mid \text{not } c$ $\quad \mid x$ $\quad \mid lit$ $\quad \mid a.f$
(integers)	$ninf ::= n \mid \infty$
(table aliases)	a

Fig. 15. The SQL grammar that results from query compilation

$$s ::= \dots \mid q$$

We write $s[b/x]$ for the substitution of b for x in s . Again we abuse the vector notation in the obvious way in order to denote simultaneous pointwise substitution. Note that min is a meta language (i.e. compiler) operation that returns the minimum of two integers, and is not part of the target language.

Proposition 1. *The database rewrite rules are strongly normalising and confluent.*

Proof. Define the size $|e|$ of an expression e as follows.

$$\begin{aligned}
 |take(n, s)| &= 1 + |s| \\
 |drop(n, s)| &= 1 + |s| \\
 |\text{for } (x \leftarrow s1) s2| &= 2 + |s1| + |s2| \\
 |\text{for } ((\bar{f} = \bar{x}) \leftarrow s1) s2| &= 1 + |s1| + |s2| \\
 |\text{let } x = b \text{ in } s| &= 1 + |s| \\
 |\text{where } (b) s| &= 1 + |s| \\
 |\text{table } t| &= 1 \\
 |[(b)]| &= 1
 \end{aligned}$$

Each rewrite rule strictly reduces the size of an expression, thus the rules are strongly normalising. The rewrite rules are orthogonal and hence weakly confluent. Confluence follows as a direct consequence of weak confluence and strong normalisation.

TABLE

```

table  $t$  with  $\bar{f}$ 
→ ( $a$  is fresh)
select  $a.\bar{f}$  from  $t$  as  $a$  where true limit  $\infty$  offset  $0$ 

```

LET

```

let  $x = b$  in  $s \rightarrow s[b/x]$ 

```

TUPLE

```

 $[(\bar{b})] \rightarrow \text{select } \bar{b} \text{ from } \bullet \text{ where true limit } \infty \text{ offset } 0$ 

```

JOIN

```

for ( $(\bar{f}=\bar{x}) <-$ 
  select  $\bar{c}_1$  from  $\bar{t}_1$  as  $\bar{a}_1$  where  $d_1$  limit  $\infty$  offset  $0$ )
  select  $\bar{c}_2$  from  $\bar{t}_2$  as  $\bar{a}_2$  where  $d_2$  limit  $\infty$  offset  $0$ 
→ ( $\bar{a}_1$  and  $\bar{a}_2$  are disjoint)
select  $\bar{c}_2[\bar{c}_1/\bar{x}]$  from  $\bar{t}_1$  as  $\bar{a}_1$ ,  $\bar{t}_2$  as  $\bar{a}_2$ 
  where  $d_1$  and  $d_2[\bar{c}_1/\bar{x}]$  limit  $\infty$  offset  $0$ 

```

WHERE

```

where ( $b$ ) (select  $\bar{c}$  from  $\bar{t}$  as  $\bar{a}$  where  $d$  limit  $m$  offset  $n$ )
→
select  $\bar{c}$  from  $\bar{t}$  as  $\bar{a}$  where  $d$  and  $b$  limit  $m$  offset  $n$ 

```

RECORD

```

for ( $z <-$  select  $\bar{c}$  from  $\bar{t}$  as  $\bar{a}$  where  $d$  limit  $m$  offset  $n$ )  $s$ 
→ ( $\bar{x}$  not free in  $s$ )
for ( $(\bar{f}=\bar{x}) <-$  select  $\bar{c}$  from  $\bar{t}$  as  $\bar{a}$ 
  where  $d$  limit  $m$  offset  $n$ )  $s[\bar{x}/z.\bar{f}]$ 

```

TAKE

```

take( $i$ , select  $Q$  limit  $m$  offset  $n$ )
→
select  $Q$  limit  $\min(m, i)$  offset  $n$ 

```

DROP

```

drop( $i$ , select  $Q$  limit  $m$  offset  $n$ )
→
select  $Q$  limit  $m-i$  offset  $n+i$ 

```

DROP $_{\infty}$

```

drop ( $i$ , select  $Q$  limit  $\infty$  offset  $n$ )
→
select  $Q$  limit  $\infty$  offset  $n+i$ 

```

Fig. 16. Database rewrite rules

5 Statically Typed Message Passing

Concurrency in Links is based on processes that send messages to mailboxes. Each process can read from a single mailbox which it owns. The message passing model is inspired by Erlang [2], but unlike in Erlang, mailboxes in Links are statically typed. To achieve this, we add a special mailbox type to the typing context, and annotate function types with the type of their mailbox.

We here present a tiny core calculus to capture the essence of process and message typing in Links. There is nothing particularly difficult here, but it does capture the essence of how our type inference works. The simple description of our core has already proven useful to researchers at MSR Cambridge working with Links.

We let A, B, C, D range over types, s, t, u range over terms, and x range over variables. A type is either the empty type $\mathbf{0}$; unit type $\mathbf{1}$; a process type $P(A)$, for a process that accepts messages of type A ; or a function type $A \rightarrow^C B$, for a function with argument of type A , result of type B , and that may accept messages of type C when evaluated. Links also supports record types and variant types (using row typing), but as those are standard we don't describe them here. Typically, a process will receive messages belonging to a variant type, and we use the empty variant type (which is isomorphic to the empty type $\mathbf{0}$) to assert that a function does not receive messages.

A typing judgement has the form $\Gamma; C \vdash t : A$, where Γ is a typing context pairing variables with types, C is the type of messages that may be accepted during execution of the term, t is the term, and A is the type of the term.

The typing rules are shown in Figure 17. The rules for variables and the constant of unit type are standard. The type of a lambda abstraction is labelled with the type C of messages that may be received by its body, where the context of the abstraction may accept messages of an unrelated type D ; while in an application, the function must receive messages of the same type C as the context in which it is applied.

If t is a term of unit type that receives messages of type C , then `spawn t` returns a fresh process identifier of type $P(C)$, where the context of the spawn may receive messages of an unrelated type D . The term `self` returns the process identifier of the current process, and so has type $P(C)$ when it appears in a context that accepts messages of type C . If s is a term yielding a process identifier of type $P(D)$ and t is a term of type D , then `send $s t$` is a term of unit type that sends the message t to the process identified by s ; it evaluates in a context that may receive messages of an unrelated type C . Finally, the term `receive` returns a value of type C when it is executed in a context that may receive messages of type C .

Links syntax differs slightly from the core calculus. The type $A \rightarrow^C B$ is written $A -\{C\} \rightarrow B$, or just $A \rightarrow B$ when C is unspecified. The core expression `self` is written as a function call `self()`, the core expression `send $t u$` is written `$t ! u$` , and the Links expression

```
receive {
  case  $p1 \rightarrow e1$ 
  ...
  case  $pn \rightarrow en$ 
}
```

$$\begin{array}{c}
\frac{}{\Gamma, x : A; C \vdash x : A} \quad \frac{}{\Gamma; C \vdash () : \mathbf{1}} \\
\frac{\Gamma, x : A; C \vdash u : B}{\Gamma; D \vdash \lambda x.u : A \rightarrow^C B} \quad \frac{\Gamma; C \vdash s : A \rightarrow^C B \quad \Gamma; C \vdash t : A}{\Gamma; C \vdash st : B} \\
\frac{\Gamma; C \vdash t : \mathbf{1}}{\Gamma; D \vdash \text{spawn } t : P(C)} \quad \frac{}{\Gamma; C \vdash \text{self} : P(C)} \\
\frac{\Gamma; C \vdash s : P(D) \quad \Gamma; C \vdash t : D}{\Gamma; C \vdash \text{send } s t : \mathbf{1}} \quad \frac{}{\Gamma; C \vdash \text{receive} : C}
\end{array}$$

Fig. 17. Statically typed message passing

$$\begin{array}{l}
\text{thread} : (A \rightarrow A) \rightarrow A \\
\text{newChannel} : \text{unit} \rightarrow \text{Chan}(A) \\
\text{put} : \text{Chan}(A) \rightarrow A \rightarrow \text{unit} \\
\text{get} : \text{Chan}(A) \rightarrow \text{unit} \rightarrow A
\end{array}$$

Fig. 18. Constants for the $\lambda(\text{fut})$ translation

corresponds to a switch on the value returned by the receive operator. (Recall that by convention the type of messages sent to a process, i.e. its mailbox type, is a variant type.)

We give a formal semantics for the typed message-passing calculus via a translation into $\lambda(\text{fut})$, Niehren et al’s concurrent extension of call-by-value λ -calculus [20]. For our purposes we need not consider all of the details of $\lambda(\text{fut})$. We just describe the image of the translation and then rely on previous results [20]. The image of the translation is simply-typed call-by-value λ -calculus extended with the constants of Figure 18.

The expression $\text{thread } (\lambda x.u)$ spawns a new process binding its *future* to the variable x in u and also returning this future. A future can be thought of as placeholder for a value that is set to contain a real value once that value becomes known. Futures are central to $\lambda(\text{fut})$, but not to our translation.

The other constants are used for manipulating asynchronous *channels*, which can be used for implementing mailboxes. The constants `newChannel`, `put` and `get` are used for creating a channel, writing to a channel, and reading from a channel. They are not built in to $\lambda(\text{fut})$, but are easily implemented in terms of $\lambda(\text{fut})$ -primitives, when $\lambda(\text{fut})$ is extended with product types. The type of channels is $\text{Chan}(A)$ ³.

We write $\Gamma \vdash^{\text{fut}} u : A$ for the $\lambda(\text{fut})$ typing judgement “ u has type A in context Γ .” The translation on terms and types is given in Figure 19. At the top level a fresh channel corresponding to the mailbox of the main process is introduced. Thereafter, the translation on terms is annotated with the current channel. Unit and variables are just mapped

³ In fact $\text{Chan}(A)$ is just the type $(A \rightarrow \text{unit}) \times (\text{unit} \rightarrow A)$ (i.e. a pair of put and get functions), but that is not relevant to our presentation.

$$\begin{aligned}
 \llbracket u \rrbracket &= \text{let } ch \text{ be } \text{newChannel} () \text{ in } \llbracket u \rrbracket^{ch} \\
 \llbracket () \rrbracket^{ch} &= () & \llbracket \text{spawn } u \rrbracket^{ch} &= \text{let } ch' \text{ be } \text{newChannel} () \text{ in} \\
 & & & \quad \text{let } _ \text{ be } \text{thread} (\lambda _ . \llbracket u \rrbracket^{ch'}) \text{ in } ch' \\
 \llbracket x \rrbracket^{ch} &= x & \llbracket \text{send } s \ t \rrbracket^{ch} &= \text{put } \llbracket s \rrbracket^{ch} \ \llbracket t \rrbracket^{ch} \\
 \llbracket \lambda x. u \rrbracket^{ch} &= \lambda x. \lambda ch'. \llbracket u \rrbracket^{ch'} & \llbracket \text{receive} \rrbracket^{ch} &= \text{get } ch \\
 \llbracket st \rrbracket^{ch} &= \llbracket s \rrbracket^{ch} \ \llbracket t \rrbracket^{ch} \ ch & \llbracket \text{self} \rrbracket^{ch} &= ch \\
 \llbracket \mathbf{1} \rrbracket &= \mathbf{1} & \llbracket P(A) \rrbracket &= \text{Chan}(\llbracket A \rrbracket) & \llbracket A \rightarrow^C B \rrbracket &= \llbracket A \rrbracket \rightarrow \llbracket P(C) \rrbracket \rightarrow \llbracket B \rrbracket
 \end{aligned}$$

Fig. 19. Translation into $\lambda(\text{fut})$

to themselves. The current mailbox is threaded through functions and applications. The spawn, send and receive constructs are mapped to thread, put and get, and self is simply mapped to the channel that corresponds to the current mailbox. The type translation maps unit to unit, process types to channel types, and function types to function types with an extra channel argument.

It is straightforward to prove by induction on typing derivations that the transformation respects typing judgements.

Proposition 2. *The transformation $\llbracket \cdot \rrbracket$ respects typing judgements.*

$$\Gamma; C \vdash u : A \quad \text{iff} \quad \llbracket \Gamma \rrbracket \vdash^{\text{fut}} \text{let } ch \llbracket C \rrbracket \text{ be } \text{newChannel} () \text{ in } \llbracket u \rrbracket^{ch} : A$$

By Proposition 2 and standard results for $\lambda(\text{fut})$ [20], we can be sure that programs that are well-typed in our message-passing calculus “do not go wrong”.

6 Issues

Links is still at an early stage of development. There are a number of shortcomings in the current design, which we plan to address. We mention some of them here.

Form abstractions. A flaw in the design of forms is that Links does not support abstraction over form components. As described in section 2.1, form interaction in Links is specified using the attributes `l:name`, which binds the value of a form field to a Links variable, and `l:onsubmit`, which contains Links code to be executed when a form is submitted. Variables bound by `l:name` are in scope within the `l:onsubmit` attribute of the lexically enclosing form. This design has several shortcomings. Firstly, it is not possible to abstract over input elements, since all input elements must be lexically contained within the form to which they belong. Secondly, it is not possible to vary the number of input elements within a form, since each input element is associated with a Links variable, and the number of variables is fixed at compile time. Thirdly, form components are not composable: it is not possible to compose input elements to create new form components which have the same interface as the language-supplied primitives, since `l:name` is only attachable to `input` XML literals.

To illustrate these problems, say we wish to construct a *date* component which we will use to allow the user to enter a number of dates into a form.

```
var date =
  <#>
    month: <input l:name="month" />
    day:   <input l:name="day" />
  </#>
```

We would then like to use instances of *date* in a form elsewhere in the program:

```
<form l:onsubmit="{e}">
  Arrival: {date}
  Departure: {date}
</form>
```

Unfortunately, Links provides no way to refer within the `l:onsubmit` attribute to the values entered into the date fields, since the input elements are not lexically enclosed within the form. The only way to write the program is to textually include the date code within the form, then refer to the individual input elements of each month and day within *e*. This violates abstraction in two ways: the component must be defined (twice!) at the point of use, and the individual elements of the component cannot be concealed from the client.

If we try to construct a form with a number of fields determined at runtime then we run into further difficulties:

```
fun (n) {
  <form l:onsubmit="{e}">{
    for (var i <- range(n))
      Field {i}: <input l:name="x" />
  }</form>
}
```

This code is also problematic: there is no way to refer to the different input fields generated in the loop, since all have the same name! Such constructs are consequently disallowed, and Links provides no way to construct forms with dynamically-varying numbers of fields.

The `l:name` mechanism provides a limited (but clean) way to refer to DOM nodes that are `<input>` elements. A related problem is that there is currently no other way of statically referring to a DOM node, which means we have to resort to calling the function `getNodeById` as illustrated in the draggable lists example.

Other systems also suffer from these problems to a greater or lesser degree. In PLT Scheme [15], composing new forms from old or making forms with a varying number of fields is possible, but not straightforward. In WASH [31] and iData [23], building forms that return new types requires declaring instances for those types in particular type classes, and in WASH a special type constructor needs to be used for forms with a varying number of fields. It was surprising for us to realize that composability of forms has received relatively little attention.

We are currently working on a design for light-weight modular form components, which uses a uniform framework to support form abstraction, varying numbers of form fields, and a mechanism that allows DOM nodes to be statically bound.

Synchronous messages. As far as we are aware, our mailbox typing scheme for Links is novel, though the translation into $\lambda(\text{fut})$ shows that it can be simulated using typed concurrent languages with support for typed channels.

Links message passing is asynchronous, but often one wants synchronous message passing, where one sends a message and waits for a result. An example where synchronous message passing is needed is the extension of the draggable lists example to support reading the content of draggable lists⁴

Although Links does not directly support synchronous message passing, it can be simulated in the same way as in Erlang. In order to send a synchronous message from process P to process Q the following sequence of events must occur:

- P sends a message (M, P) to Q ,
- P blocks waiting for a reply $\text{Reply}(N)$,
- on receiving the message (M, P) the process Q sends a message $\text{Reply}(N)$ to P ,
- P receives the reply message $\text{Reply}(N)$.

However, Erlang is untyped; in Links, the simulation of synchronous messages has the unfortunate side-effect of polluting the type of process P 's mailbox, to include a different constructor for each distinct type of reply message associated with synchronous messages sent by P .

Type pollution of this kind makes it hard to structure programs cleanly. We are considering moving to a different model of concurrency that makes it easier to support typed synchronous message passing, such as the Join calculus [10], or even removing explicit concurrency from the language altogether and replacing it with something like functional reactive programming [34].

Database abstraction. As we showed in Section 4, we can guarantee to compile a useful fragment of Links into SQL. However, this fragment still lacks many important features. We do not yet deal with aggregate functions such as `count`, `sum`, and `average`, and we only support `order by` attached to a single loop iterator (rather than a nest of loop iterators). One can express the equivalent of a `group by` construct as a nested loop, but we do not yet translate code into `group by` when appropriate, nor yet support any other form of nested queries.

We also do not yet provide adequate support for abstracting over a query. For instance, suppose we try to abstract over the condition in the Dictionary Suggest example. We can amend the `completions` function to take a condition encoded as a function from unit to boolean in place of the prefix.

```
fun completions(cond) server {
  if (s == "") [] else {
    take(10, for (var def <-- defsTable)
      where (cond()) orderby (def.word)
        [def])
  }
}
```

⁴ See <http://groups.inf.ed.ac.uk/links/examples/>

Only once the condition is known is it possible to compile this to efficient SQL. The current compiler produces spectacularly inefficient SQL, which returns the entire dictionary, leaving the Links interpreter to then filter according to the condition and then take the first 10 elements of the filtered list. To produce adequately efficient code, it is necessary to inline calls to *completions* with the given predicate. But our compiler does not yet perform inlining, and in general inlining can be difficult in the presence of separate compilation.

Some of the problems outlined above are easy. For instance, we should be able to support aggregates in the same style that we support `take` and `drop`; and it is straightforward to extend our techniques to support `orderby` on nested loops. However, other problems are more challenging; in particular, it is not immediately clear how to extend the fragment described in Section 4 to support abstraction over predicates usefully.

Microsoft's Linq addresses all these issues, but does so by (a) requiring the user to write code that closely resembles SQL and (b) a subtle use of the type system to distinguish between a procedure that returns a value and a procedure that returns code that generates that value. It remains an open research problem to determine to what extent one can support efficient compilation into SQL without introducing separate syntax and semantics for queries as is done in Linq.

7 Conclusion

We have demonstrated that it is possible to design a single language in which one can program all three tiers of a web application. This eliminates the usual impedance mismatch and other problems connected with writing an application distributed across multiple languages and computers. We have shown that we can support Ajax-style interaction, programming applications such as Dictionary Suggest and draggable lists. The language compiles to JavaScript to run in the client, and SQL to run on the database.

Ultimately, we would like to grow a user community for Links. This depends on a number of factors, many of which are not well understood. One important factor seems to be the availability of good libraries. Good interfaces to other systems may help to rapidly develop useful functionality. A vexing question here is how to access facilities of languages that take a mostly imperative view of the world without vitiating the mostly functional approach taken by Links.

Already, researchers at Microsoft Cambridge and the University of Maryland have begun projects to add security features to Links; and one paper has been published that uses a modified version of Links to enhance security [32].

To make Links truly successful will require bringing together researchers and developers from many localities, many perspectives, and many communities. We would like to join our efforts with those of others — let us know if you are interested!

References

1. Atkins, D.L., Ball, T., Bruns, G., Cox, K.C.: Mawl: A domain-specific language for form-based services. *Software Engineering* 25(3), 334–346 (1999)
2. Armstrong, J.: Concurrency oriented programming in Erlang. Invited talk, FFG (2003)

3. Balat, V.: Ocsigen: typing web interaction with objective Caml. In: Proceedings of the 2006 workshop on ML, Portland, Oregon (September 2006)
4. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C#. *TOPLAS* 26(5) (2004)
5. Benton, N., Kennedy, A., Russo, C.: Adventures in interoperability: the SML .NET experience. *PPDP* (2004)
6. Bierman, G., Meijer, E., Schulte, W.: Programming with rectangles, triangles, and circles. In: *XML Conference* (2003)
7. Buneman, P., Naqvi, S., Tannen, V., Wong, L.: Principles of programming with complex objects and collection types. *TCS* 149(1) (1995)
8. Burstall, R., MacQueen, D., Sannella, D.: Hope: An experimental applicative language. In: *Lisp Conference* (1980)
9. El-Ansary, S., Grolaux, D., Van Roy, P., Rafea, M.: Overcoming the multiplicity of languages and technologies for web-based development. In: Van Roy, P. (ed.) *MOZ 2004*. LNCS, vol. 3389, Springer, Heidelberg (2005)
10. Fournet, C., Gonthier, G.: The Join Calculus: a language for distributed mobile programming. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) *APPSEM 2000*. LNCS, vol. 2395, Springer, Heidelberg (2002)
11. Garret, J.: *Ajax: a new approach to web applications* (2005)
12. Graham, P.: Method for client-server communications through a minimal interface. United States Patent no. 6,205,469 (March 20, 2001)
13. Graham, P.: *Beating the averages* (2001)
14. Graunke, P., Findler, R.B., Krishnamurthi, S., Felleisen, M.: Automatically restructuring programs for the web. *ASE* (2001)
15. Graunke, P., Krishnamurthi, S., van der Hoeven, S., Felleisen, M.: Programming the web with high-level programming languages. In: Sands, D. (ed.) *ESOP 2001 and ETAPS 2001*. LNCS, vol. 2028, Springer, Heidelberg (2001)
16. Gapeyev, V., Levin, M., Pierce, B., Schmitt, A.: *The Xtatic experience*. *PLAN-X* (2005)
17. labs.google.com/suggest
18. Microsoft Corporation. *DLinq: .NET Language Integrated Query for Relational Data* (September 2005)
19. Møller, A., Schwartzbach, M.: The design space of type checkers for XML transformation languages. In: Eiter, T., Libkin, L. (eds.) *ICDT 2005*. LNCS, vol. 3363, Springer, Heidelberg (2004)
20. Niehren, J., Schwinghammer, J., Smolka, G.: A Concurrent Lambda Calculus with Futures. *TCS*, 364(3) (2006)
21. Narra, G.: *ObjectGraph Dictionary*, <http://www.objectgraph.com/dictionary/how.html>
22. Odersky, M., et al.: *An overview of the Scala programming language*. Technical report, EPFL Lausanne (2004)
23. Plasmeijer, R., Achten, P.: *iData For The World Wide Web: Programming Interconnected Web Forms*. In: Hagiya, M., Wadler, P. (eds.) *FLOPS 2006*. LNCS, vol. 3945, Springer, Heidelberg (2006)
24. Pottier, F., Rémy, D.: The essence of ML type inference. In: Pierce, B. (ed.) *Advanced Topics in Types and Programming Languages*, ch. 10, pp. 389–489. MIT Press, Cambridge (2005)
25. Queinnec, C.: *Continuations to program web servers*. *ICFP* (2000)
26. Queinnec, C.: *Inverting back the inversion of control or, continuations versus page-centric programming*. *SIGPLAN Not* (2003)
27. Reynolds, J.: *Definitional interpreters for higher-order programming languages*. In: *ACM '72: Proceedings of the ACM annual conference* (1972)

28. Ruby on Rails, <http://www.rubyonrails.org/>
29. Serrano, M., Galesio, E., Loitsch, F.: HOP, a language for programming the Web 2.0. In: Proceedings of the First Dynamic Languages Symposium, Portland, Oregon (October 2006)
30. Syme, D.: F#r web page, research.microsoft.com/projects/ilx/fsharp.aspx
31. Thiemann, P.: WASH/CGI: server-side web scripting with sessions and typed, compositional forms. In: Krishnamurthi, S., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257, Springer, Heidelberg (2002)
32. Trevor, J., Swamy, N., Hicks, M.: Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. World Wide Web (May 2007)
33. Van Roy, P.: Convergence in language design: a case of lightning striking four times in the same place. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, Springer, Heidelberg (2006)
34. Wan, Z., Hudak, P.: Functional reactive programming from first principles. In: PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, Vancouver, British Columbia, Canada (2000)
35. Wiederman, B., Cook, W.: Extracting queries by static analysis of transparent persistence. POPL (2007)
36. Wong, L.: Kleisli, a functional query system. JFP, 10(1) (2000)
37. XML Query and XSL Working Groups. XQuery 1.0: An XML Query Language, W3C Working Draft (2005)

Author Index

- Agha, Gul 246
Ahrendt, Wolfgang 70
Aranda, Jesús 185
Artho, Cyrille 26

Barthe, Gilles 152
Beckert, Bernhard 70
Biere, Armin 26
Burdy, Lilian 152

Charles, Julien 152
Cooper, Ezra 266

Dezani-Ciancaglini, Mariangiola 207
Di Giusto, Cinzia 185
Donkervoet, Bill 246
Drossopoulou, Sophia 207

Frantzen, Lars 1

Giachino, Elena 207
Grégoire, Benjamin 152

Hähnle, Reiner 70
Honiden, Shinichi 26
Huisman, Marieke 152

Jeannet, Bertrand 47
Jéron, Thierry 47

Krukow, Karl 175

Lanet, Jean-Louis 152
Leucker, Martin 127
Lindley, Sam 266

Nielsen, Mogens 175

Palamidessi, Catuscia 185
Pavlova, Mariela 152

Requet, Antoine 152
Rümmer, Philipp 70
Rusu, Vlad 47

Sassone, Vladimiro 175
Schmitt, Peter H. 70
Sirjani, Marjan 102

Tretmans, Jan 1

Valencia, Frank D. 185

Wadler, Philip 266

Yallop, Jeremy 266
Yoshida, Nobuko 207