Lecture Notes in Computer Science 5051

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

Gilles Barthe Frank S. de Boer (Eds.)

Formal Methods for Open Object-Based Distributed Systems

10th IFIP WG 6.1 International Conference, FMOODS 2008
Oslo, Norway, June 4-6, 2008
Proceedings

Volume Editors

Gilles Barthe
INRIA Sophia-Antipolis Méditerranée
2004 route des lucioles - BP 93 06902 Sophia Antipolis Cedex
France
E-mail: gilles.barthe@inria.fr

Frank S. de Boer
Centrum voor Wiskunde en Informatica (CWI)
Kruislaan 413, 1090 GB Amsterdam
The Netherlands
E-mail: frb@cwi.nl

Library of Congress Control Number: 2008928586

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-68862-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-68862-4 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12278145 06/3180 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 10th IFIP Working Group 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2008). The conference was part of the Third Federated conferences on Distributed Computing Techniques (DisCoTec), together with the 10th International Conference on Coordination Models and Languages (COORDINATION 2008) and the 8th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2008). We are grateful to Frank Eliassen and Einar Broch Johnsen of the University of Oslo for the excellent organization of this event in Olso, Norway, June 4–6, 2008.

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

- Formal methods
- Distributed systems
- Object-based technology

The 14 papers presented at FMOODS 2008 and included in this volume were selected by the Program Committee among 35 submissions. Each submission was reviewed by at least three Program Committee members. They all reflect the scope of the conference and cover the following topics: semantics of object-oriented programming; formal techniques for specification, analysis, and refinement; model checking; theorem proving and deductive verification; type systems and behavioral typing; formal methods for service-oriented computing; integration of quality of service requirements into formal models; formal approaches to component-based design; and applications of formal methods.

The invited speaker of FMOODS 2008, Andrew Myers from Cornell University (USA), presented his work on guiding distributed systems synthesis with language-based security policies. In his talk he described a higher-level approach to programming secure systems. Andrew detailed two applications of his approach: building secure Web applications using partitioning between clients and servers, and building more general secure systems by synthesizing fault-tolerance protocols for availability.

Finally, we thank all authors for the high quality of their contributions, and the Program Committee members and the external reviewers for their help in selecting the papers for this volume.

The use of the EasyChair conference system was very helpful for organizing the technical program and proceedings of FMOODS 2008. Thanks to Nathalie Bellesso for her help in preparing the proceedings.

June 2008

Gilles Barthe
Frank de Boer

Organization

General Chairs

Frank Eliassen, University of Oslo
Einar Broch Johnsen, University of Oslo

Program Chairs

Gilles Barthe, INRIA Sophia Antipolis Méditerrane
Frank de Boer, CWI, The Netherlands

Steering Committee

Marcello Bonsangue (University of Leiden)
Einar Broch Johnsen (University of Oslo)
John Derrick (University of Sheffield)
Roberto Gorrieri (University of Bologna)
Elie Najm (University of Oslo)
Carolyn Talcott (SRI International)
Heike Wehrheim (University of Paderborn)
Gianluigi Zavattaro (University of Bologna)

Program Committee

Bernhard K. Aichernig (Technical University of Graz)
Gilles Barthe (INRIA Sophia Antipolis Méditerrane)
Frank de Boer (CWI, The Netherlands)
Marcello Bonsangue (University of Leiden)
Paulo Borba (Federal University of Pernambuco)
Einar Broch Johnsen (University of Oslo)
Dave Clarke (CWI)
John Derrick (University of Sheffield)
Sophia Drossopoulou (Imperial College London)
Seif Haridi (SICS/KTH)
Reiner Haehnle (Chalmers University of Technology)
John Hatcliff (Kansas State University)
Peter Gorm Larsen (Engineering College of Aarhus)
Antonia Lopes (University of Lisbon)
Peter Mueller (ETH/Microsoft)
Elie Najm (ENST, Paris)
David Naumann (Stevens Institute of Technology)

Uwe Nestmann (Technical University of Berlin)
Frank Piessens (Katholieke Universiteit Leuven)
Arnd Poetzsch-Heffter (University of Kaiserslautern)
Antonio Ravara (Technical University Lisbon)
Arend Rensink (University of Twente)
Grigore Rosu (University of Illinois at Urbana-Champaign)
Martin Steffen (University of Oslo)
Carolyn Talcott (SRI International)
Heike Wehrheim (University of Paderborn)
Elena Zucca (University of Genova)

Referees

Erika Abraham
Davide Ancona
Eduardo Aranha
Johannes Borgström
Harald Brandl
Walter Cazzola
Maura Cerioli
Feng Chen
Jean-Marie Gaillourdet
Kathrin Geilmann
Rohit Gheyi
Andreas Griesmayer
Andreas Gruener
Juliano Iyoda
Bart Jacobs
Vladimir Klebanov
Patrick Meredith
Björn Metzler
Cristian Prisacariu
Gianna Reggio
Philipp Ruemmer
Rudolf Schlatte
Gerardo Schneider
Jan Schäfer
Jan Smans
Christian Soltenborn
Gheorghe Stefanescu
Peter Ölveczky
Marcelo d'Amorim

Table of Contents

Formal Methods for Open Object-Based Distributed Systems

Invited Talk

Accepted Papers

Guiding Distributed Systems Synthesis with Language-Based Security Policies

Andrew Myers

Cornell University, USA

Abstract. The distributed information systems we use every day are becoming more complex and interconnected. Can we trust them with our information? Currently there is no good way to check that distributed software uses information securely, even if we have the source code. Many mechanisms are available, but are error-prone: for example, encryption, various cryptographic protocols, access control, and replication. But it is hard to know when we are using these mechanisms in a way that correctly enforces application security requirements.

This talk describes a higher-level approach to programming secure systems. Instead of using security mechanisms directly, the programming language incorporates explicit security policies specifying the confidentiality, integrity, and availability of information. The compiler then automatically transforms the source code to run securely on the available host machines, and uses a variety of security mechanisms in order to satisfy security policies. The result is systems that are secure by construction. We look at two applications of this approach: building secure web applications using partitioning beween clients and servers, and building more general secure systems by synthesizing fault-tolerance protocols for availability.

Joint work with Steve Chong, Jed Liu, Nate Nystrom, Xin Qi, K. Vikram, Steve Zdancewic, Lantian Zheng, and Xin Zheng.

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, p. 1, 2008.

Termination Analysis of Java Bytecode

Elvira Albert[1], Puri Arenas[1], Michael Codish[2],
Samir Genaim[3], Germán Puebla[3], and Damiano Zanardini[3]

[1] DSIC, Complutense University of Madrid (UCM), Spain
[2] CS, Ben-Gurion University of the Negev, Israel
[3] CLIP, Technical University of Madrid (UPM), Spain

Abstract. Termination analysis has received considerable attention, traditionally in the context of declarative programming, and recently also for imperative languages. In existing approaches, termination is performed on source programs. However, there are many situations, including mobile code, where only the compiled code is available. In this work we present an automatic termination analysis for sequential Java Bytecode programs. Such analysis presents all of the challenges of analyzing a low-level language as well as those introduced by object-oriented languages. Interestingly, given a bytecode program, we produce a *constraint logic* program, CLP, whose termination entails termination of the bytecode program. This allows applying the large body of work in termination of CLP programs to termination of Java bytecode. A prototype analyzer is described and initial experimentation is reported.

1 Introduction

It has been known since the pre-computer era that it is not possible to write a program which correctly decides, in all cases, if another program will *terminate*. However, termination analysis tools strive to find proofs of termination for as wide a class of (terminating) programs as possible. Automated techniques are typically based on analyses which track *size* information, such as the value of numeric data or array indexes, or the size of data structures. This information is used for specifying a *ranking function* which strictly decreases on a well-founded domain on each computation step, thus guaranteeing termination.

In the last two decades, a variety of sophisticated termination analysis tools have been developed, primarily for less-widely used programming languages. These include analyzers for term rewrite systems [15], and logic and functional languages [18,10,17]. Termination-proving techniques are also emerging in the imperative paradigm [6,11,15], even for dealing with large industrial code [11].

Static analysis of *Java ByteCode* (JBC for short) has received considerable attention lately [25,23,24,22,1]. The present paper presents a static analysis for sequential JBC which is, to the best of our knowledge, the first approach to proving termination. Bytecode is a low-level representation of a program, designed to be executed by a virtual machine rather than by dedicated hardware. As such, it is usually higher level than actual machine code, and independent of

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 2–18, 2008.

the specific hardware. This, together with its security features, makes JBC [19] the chosen language for many *mobile code* applications. In this context, analysis of JBC programs may enable performing a certain degree of static (i.e., before execution) verification on program components obtained from untrusted providers. *Proof-Carrying Code* [20] is a promising approach in this area: mobile code is equipped with certain *verifiable evidence* which allows deployers to independently verify properties of interest about the code. Termination analysis is also important since the verification of functional program properties is often split into separately proving partial correctness and termination.

Object-oriented languages in general, and their low-level (bytecode) counterparts in particular, present new challenges to termination analyzers: (1) loops originate from different sources, such as conditional and unconditional jumps, method calls, or even exceptions; (2) *size measures* must consider primitive types, user defined objects, and arrays; and (3) *tracking* data is more difficult, as data can be stored in variables, operand stack elements or heap locations.

Analyzing JBC is a necessity in many situations, including mobile code, where the user only has access to compiled code. Furthermore, it can be argued that analyzing low-level programs can have several advantages over analyzing their high-level (Java) counterparts. One advantage is that low-level languages typically remain stable, as their high-level counterparts continue to evolve — analyzers for bytecode programs need not be enhanced each time a new language construct is introduced. Another advantage is that analyzing low-level code narrows the gap between what is verified and what is actually executed. This is relevant, for example, in safety critical applications.

In this paper we take a semantic-based approach to termination analysis, based on two steps. The first step transforms the bytecode into a *rule-based* program where all loops and all variables are represented uniformly, and which is semantically equivalent to the bytecode. This rule-based representation is based on previous work [1] in cost analysis, and is presented in Sec. 2. In the second step (Sec. 3), we adapt directly to the rule-based program standard techniques which usually prove termination of high-level languages. Sec. 4 reports on our prototype implementation and validates it by proving termination of a series of object-oriented benchmarks, containing recursion, nested loops and data structures such as trees and arrays. Conclusions and related work are presented in Sec. 5.

2 Java Bytecode and Its Rule-Based Representation

We consider a subset of the Java Virtual Machine (JVM) language which handles integers and object creation and manipulation (by accessing fields and calling methods). For simplicity, exceptions, arrays, interfaces, and primitive types besides integers are omitted. Yet, these features can be easily handled within our setting: all of them are implemented in our prototype and included in benchmarks in Table 1. A full description of the JVM [19] is out of the scope of this paper.

A sequential JBC program consists of a set of *class files*, one for each class, partially ordered with respect to the subclass relation $\preceq$. A class file contains

information about its name, the class it extends, and the fields and methods it defines. Each method has a unique signature m from which we can obtain the class, denoted $class(m)$, where the method is defined, the name of the method, and its signature. When it is clear from the context, we ignore the class and the types parts of the signature. The bytecode associated with m is a sequence of bytecode instructions $\langle pc_1{:}b_1, \ldots, pc_n{:}b_n \rangle$, where each b_i is a *bytecode instruction*, and pc_i is its address. The local variables of a method are denoted by $\langle l_0, \ldots, l_{n-1} \rangle$, of which the first $k \leq n$ are the formal parameters, and l_0 corresponds to the *this* reference (unlike Java, in JBC, the *this* reference is explicit). Similarly, each field f has a unique signature, from which we can obtain its name and the name of the class it belongs to. The bytecode instructions we consider include:

$$
\begin{array}{rcl}
bcInst & ::= & \mathsf{istore}\ v \mid \mathsf{astore}\ v \mid \mathsf{iload}\ v \mid \mathsf{aload}\ v \mid \mathsf{iconst}\ i \mid \mathsf{aconst\_null} \\
& \mid & \mathsf{iadd} \mid \mathsf{isub} \mid \mathsf{iinc}\ v\ n \mid \mathsf{imul} \mid \mathsf{idiv} \\
& \mid & \mathsf{if\_}\phi\ pc \mid \mathsf{goto}\ pc \mid \mathsf{ireturn} \mid \mathsf{areturn} \\
& \mid & \mathsf{new}\ c \mid \mathsf{invokevirtual}\ m \mid \mathsf{invokespecial}\ m \mid \mathsf{getfield}\ f \mid \mathsf{putfield}\ f
\end{array}
$$

where c is a class, ϕ is a comparison condition on numbers (ne, le, icmpgt) or references (null, nonnull), v is a local variable, i is an integer, and pc is an instruction address. Briefly, instructions are: (row 1) stack operations referring to constants and local variables; (row 2) arithmetic operations; (row 3) jumps and method return; and (row 4) object-oriented instructions. All instructions in row 3, together with invokevirtual, are *branching* (the others are *sequential*). For simplicity, we will assume all methods to return a value. Fig. 1 depicts the bytecode for the iterative method *fact*, where indexes $0, \ldots, 3$ stands for local variables *this*, n, *ft* and i respectively. $next(pc)$ is the address immediately after the program counter pc. As instructions have different sizes, addresses do not always increase by one (e.g., $next(6){=}9$).

We assume an operational semantics which is a subset of the JVM specification [19]. The execution environment of a bytecode program consists of a *heap* h and a stack A of *activation records*. Each activation record contains a program counter, a local operand stack, and local variables. The heap contains all objects (and arrays) allocated in the memory. Each method invocation generates a new activation record according to its signature. Different activation records do not share information, but may contain references to the same object in the heap.

2.1 From Bytecode to Control Flow Graphs

The JVM language is unstructured. It allows conditional and unconditional jumps as well as other implicit sources of branching, such as virtual method invocation and exception throwing. The notion of a *Control Flow Graph* (CFG for short) is a well-known instrument which facilitates reasoning about programs in unstructured languages. A CFG is similar to the older notion of a flow chart, but CFGs include a concept of "call to" and "return from". Methods in the bytecode program are represented as CFGs, and calls from one method to another correspond to calls between these graphs. In order to build CFGs, the first step

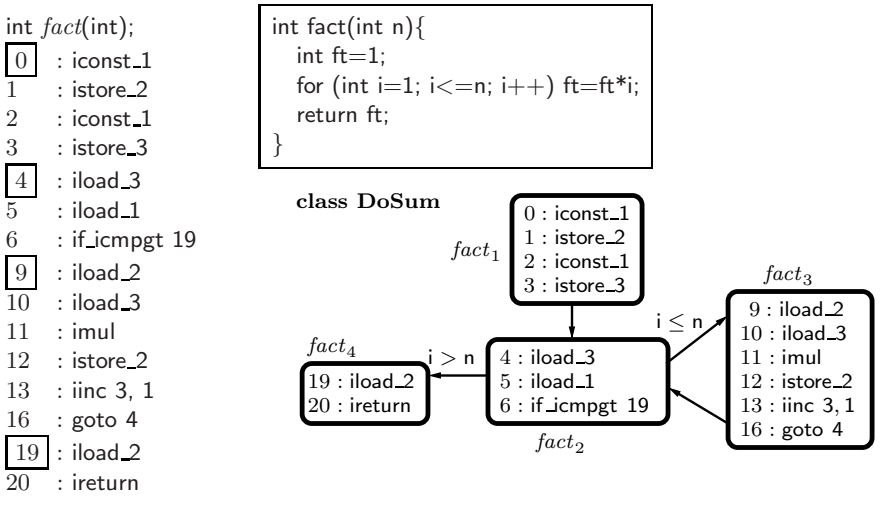


Fig. 1. A JBC method (left) with its corresponding source (center) and its CFG (right)

is to partition a sequence of bytecode instructions into a set of maximal subsequences, or basic blocks, of instructions which execute sequentially, i.e., without branching nor jumping. Given a bytecode instruction $pc{:}b$, we say that $pc'{:}b'$ is a *predecessor* of $pc{:}bc$ if one of the following conditions holds: (1) b'=goto pc, (2) b'=if_ϕ pc, (3) $next(pc')$=pc.

Definition 1 (partition to basic blocks). *Given a method m and its sequence of bytecode instructions $\langle pc_1{:}b_1, \ldots, pc_n{:}b_n\rangle$, a partition into basic blocks $m_1, \ldots, m_k$ takes the form*

$$\underbrace{pc_{i_1}{:}b_{i_1}, \ldots, pc_{f_1}{:}b_{f_1}}_{m_1}, \underbrace{pc_{i_2}{:}b_{i_2} \ldots, pc_{f_2}{:}b_{f_2}}_{m_2}, \ldots \underbrace{pc_{i_k}{:}b_{i_k} \ldots, pc_{f_k}{:}b_{f_k}}_{m_k}$$

where i_1=1, f_k=n and

1. *the number of basic blocks (i.e. k) is minimal;*
2. *in each basic block m_j, only the instruction b_{f_j} can be branching; and*
3. *in each basic block m_j, only the instruction b_{i_j} can have more than one predecessor.*

A partition to basic blocks can be obtained as follows: the first sequence m_1 starts at pc_1 and ends at pc_{f_1}=$min(pc_{e_1}, pc_{s_1})$, where pc_{e_1} is the address of the first branching instruction after pc_1, and pc_{s_1} is the first address after pc_1 s.t. the instruction at address $next(pc_{s_1})$ has more than one predecessor. The sequence m_2 is computed similarly starting at pc_{i_2}=$next(pc_{f_1})$, etc. Note that this partition can be computed in two passes: the first computes the predecessors, and the second defines the beginning and end of each sub-sequence.

Example 1. The JBC *fact* method on the left of Fig. 1 is partitioned into four basic blocks. The initial addresses (pc_{i_x}) of these blocks are shown within boxes. Each block is labeled by $fact_{id}$ where id is a unique block identifier. A directed edge indicates a control flow from the last instruction in the source node to the first instruction in the destination node. Edges may be labeled by a guard which states conditions under which the edge may be traversed during execution. □

In *invokevirtual*, due to dynamic dispatching, the actual method to be called may not be known at compile time. To facilitate termination analysis, we capture this information and introduce it explicitly in the CFG. This is done by adding new blocks, the *implicit basic blocks*, containing calls to *actual methods* which might be called at runtime. Moreover, access to these blocks is guarded by mutually exclusive conditions on the runtime class of the calling object.

Definition 2 (implicit basic block). *Let m be a method which contains an instruction of the form pc:b, where b=*invokevirtual *m'. Let M be a superset of the methods (signatures) that might actually be called at runtime when executing pc:b. The implicit basic block for $m''{\in}M$ is $m_{pc:c}$, where $c{=}class(m'')$ if $class(m''){\preceq}class(m')$, otherwise $c{=}class(m')$. The block includes the single special instruction* invoke(m'). *The guard of $m_{pc:c}$ is $m^g_{pc:c}{=}$*instanceof(n, c, D), *where $D{=}\{class(m'') \mid m''{\in}M,\ class(m''){\prec}c\}$, and n is the arity of m'.*

It can be seen that m is used to denote both methods and blocks in order to make them globally unique. The above condition instanceof(n, c, D) states that the $(n{+}1)$th stack element (from the top) is an instance of class c and not an instance of any class in D. Computing the set M in the above definition can be statically done by considering the class hierarchy and the method signature, which is clearly a safe approximation of the set of the actual methods that might be called at runtime when executing b. However, in some cases this might result in a much larger set than the actual one, which in turn affects the precision and performance of the corresponding static analysis. In such cases, class analysis [25] is usually applied to reduce this set as it gives information about the possible runtime classes of the object whose method is being called. Note that the instruction invoke(m') does not appear in the original bytecode, but it is instrumental to define our rule-based representation in Sec. 2.2. For example, consider the CFG in Fig. 2, which corresponds to the recursive method *doSum* and calls *fact*. This CFG contains two implicit blocks labeled $doSum_{11:DoSum}$ and $doSum_{19:DoSum}$.

The following definition formalizes the notion of a CFG for a method. Although the invokespecial bytecode instruction always corresponds to only one possible method call which can be identified from the symbolic method reference, in order to simplify the presentation, we treat it as invokevirtual, and associate it to a single implicit basic block with the *true* guard. Note that every bytecode instruction belongs to exactly one basic block. By $BlockId(pc, m){=}i$ we denote the fact that the instruction pc in m belongs to block m_i. In addition, for a given *invokevirtual* instruction pc:b in a method m, we use M^m_{pc} and G^m_{pc} to denote the set of its *implicit basic blocks* and their corresponding guards respectively.

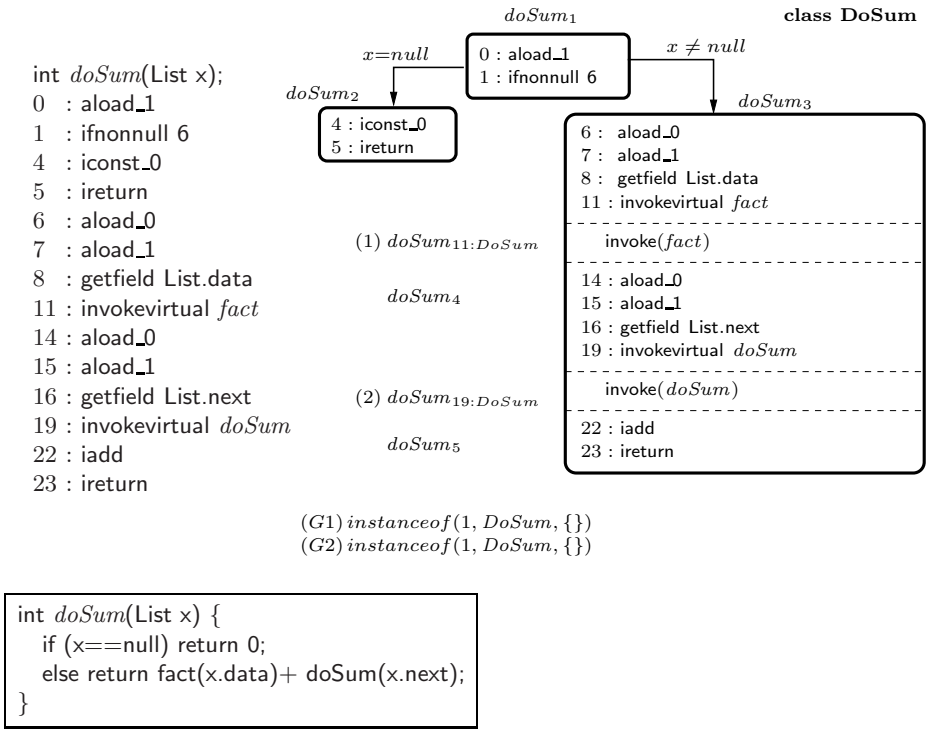


Fig. 2. The Control Flow Graph of the *doSum* example

Definition 3 (CFG). *The* control flow graph *for a method m is a graph $G = \langle \mathcal{N}, \mathcal{E} \rangle$. Nodes $\mathcal{N}$ consist of:*

(a) basic blocks $m_1, \ldots, m_k$ of m; and
(b) implicit basic blocks corresponding to calls to methods.

Edges in $\mathcal{E}$ take the form $\langle m_i \rightarrow m_j, condition_{ij} \rangle$ where m_i and m_j are, resp., the source *and* destination *node, and $condition_{ij}$ is the Boolean condition labeling this transition. The set of edges is constructed, by considering each node $m_i \in \mathcal{N}$ which corresponds to a (non-implicit) basic block, whose last instruction is denoted as pc:b, as follows:*

1. *if b=*goto *pc′ and j=$BlockId(pc', m)$ then we add $\langle m_i \rightarrow m_j, true \rangle$ to $\mathcal{E}$;*
2. *if b=*if_ϕ *pc′, j=$BlockId(pc', m)$ and i'=$BlockId(next(pc), m)$ then we add both $\langle m_i \rightarrow m_j, \phi \rangle$ and $\langle m_i \rightarrow m_{i'}, \neg\phi \rangle$ to $\mathcal{E}$;*
3. *if b∈{*invokevirtual *m',* invokespecial *m'}, and i'=$BlockId(next(pc), m)$ then, for all $d \in M^m_{pc}$ and its corresponding $g^m_{pc:d} \in G^m_{pc}$, we add $\langle m_i \rightarrow m_{pc:d}, g^m_{pc:d} \rangle$ and $\langle m_{pc:d} \rightarrow m_{i'}, true \rangle$ to $\mathcal{E}$;*
4. *otherwise, if j=$BlockId(next(pc), m)$ then we add $\langle m_i \rightarrow m_j, true \rangle$ to $\mathcal{E}$.*

For conciseness, when a branching instruction b involving implicit blocks leads to a single successor block, we include the corresponding invoke instruction within the basic block b belongs to. For instance, consider that the classes *DoSum* and *List* are not extended by any other class. In this case, the branching instructions 11 and 19 have a single continuation. Their associated implicit blocks marked with (1) and (2) in Fig. 2 are, thus, just included within the basic block $doSum_3$. $G1$ and $G2$ at the bottom indicate the guards which should label the edge.

2.2 Rule-Based Representation

The CFG, while having advantages, is not optimal for our purposes. Therefore, we introduce a *Rule-Based Representation* (RBR) on which we demonstrate our approach to termination analysis. This RBR is based on a recursive representation presented in previous work [1], where it has been used for cost analysis.

The main advantages of the RBR are that: (1) all iterative constructs (loops) fit in the same setting, independently of whether they originate from recursive calls or iterative loops (conditional and unconditional jumps); and (2) all variables in the local scope of the method a block corresponds to (formal parameters, local variables, and stack values) are represented uniformly as explicit arguments. This is possible as in JBC the height of the *operand stack* at each program point is statically known. We prefer to use this rule-based representation, rather than other existing ones (e.g., BoogiePL [13] or those in Soot [26]), as in a simple post-processing phase we can eliminate almost all stack variables, which results, as we will see in Sec. 3.1, in a more efficient analysis.

A *Rule-Based Program* (RBP for short) defines a set of *procedures*, each of them defined by one or more rules. As we will see later, each block in the CFG generates one or two procedures. Each rule has the form $head(\bar{x}, \bar{y}) := guard, instr, cont$ where $head$ is the name of the procedure the rule belongs to, $\bar{x}$ and $\bar{y}$ indicate sequences $\langle x_1, \ldots, x_n \rangle$, $n>0$ (resp. $\langle y_1, \ldots, y_k \rangle$, $k>0$) of input (resp. output) arguments, $guard$ is of the form $\mathsf{guard}(\phi)$, where ϕ is a Boolean condition on the variables in $\bar{x}$, $instr$ is a sequence of (decorated) bytecode instructions, and $cont$ indicates a possible call to another procedure representing the continuation of this procedure. In principle, $\bar{x}$ includes the method's local variables and the stack elements at the beginning of the block. In most cases, $\bar{y}$ only needs to store the return value of the method, which we denote by r. For simplicity, guards of the form $\mathsf{guard}(true)$ are omitted. When a procedure p is defined by means of several rules, the corresponding guards must cover all cases and be pairwise exclusive.

Decorating Bytecode Instructions. In order to make all arguments explicit, each bytecode instruction in $instr$ is *decorated* explicitly with the (local and stack) variables it operates on. We denote by $t{=}stack\_height(pc, m)$ the height of the stack immediately before the program point pc in a method m. Function dec in the following table shows how to *decorate* some selected instructions, where n is the number of arguments of m.

pc:b	dec(b)
iconst i	$iconst(i, s_{t+1})$
istore v	$istore(s_t, \ell_v)$
iload v	$iload(l_v, s_{t+1})$
new c	$new(c, s_{t+1})$
ireturn	$ireturn(s_t, r)$

pc:b	dec(b)
iadd	$iadd(s_{t-1}, s_t, s_{t-1})$
invoke(m)	$m(\langle s_{t-n}, \ldots, s_t\rangle, \langle s_{t-n}\rangle)$
getfield f	$getfield(f, s_t, s_t)$
putfield f	$putfield(f, s_{t-1}, s_t, s_{t-1})$
guard($icmpgt$)	guard($icmpgt(s_{t-1}, s_t)$)

Guards are translated according to the bytecode instruction they come from. Note that branching instructions do not need to appear in the RBR, since their effect is already captured by the branching at the RBR level and since invoke instructions are replaced by calls to the entry rule of the corresponding method.

Definition 4 (RBR). *Let m be a method with $l_0, \ldots, l_{n-1}$ local variables, of which $l_0, \ldots, l_{k-1}$ are the formal parameters together with the* this *reference l_0 ($k \leq n$), and let $\langle \mathcal{N}, \mathcal{E}\rangle$ be its CFG. The* rule-based representation *of $\langle \mathcal{N}, \mathcal{E}\rangle$ is* $\mathsf{rules}(\langle \mathcal{N}, \mathcal{E}\rangle) = \mathsf{entry}(\langle \mathcal{N}, \mathcal{E}\rangle) \bigcup_{m_p \in \mathcal{N}} \mathsf{translate}(m_p, \langle \mathcal{N}, \mathcal{E}\rangle)$*, with:*

$$\mathsf{entry}(\langle \mathcal{N}, \mathcal{E}\rangle) = \{m(\langle \ell_0, \ldots, \ell_{k-1}\rangle, \langle r\rangle) := \mathsf{init\_local\_vars}(\langle l_k, \ldots, l_{n-1}\rangle), m_1(\langle \ell_0, \ldots, \ell_{n-1}\rangle, \langle r\rangle)\}$$

where the call init_local_vars *initializes the local variables of the method, and*

$$\mathsf{translate}(m_p, \langle \mathcal{N}, \mathcal{E}\rangle) = \begin{cases} \{m_p(\langle \bar{l}, s_0, \ldots, s_{p_i-1}\rangle, \langle r\rangle) := TBC^p_m.\} & \nexists \langle m_p \mapsto \_, \_\rangle \in \mathcal{E} \\ \{m_p(\langle \bar{l}, s_0, \ldots, s_{p_i-1}\rangle, \langle r\rangle) := TBC^p_m, m^c_p(\langle \bar{l}, s_0, \ldots, s_{p_o-1}\rangle, \langle r\rangle).\} \bigcup & \\ \quad \{m^c_p(\langle \bar{l}, s_0, \ldots, s_{p_o-1}\rangle, \langle r\rangle) := g, m_q(\langle \bar{l}, s_0, \ldots, s_{q_i-1}\rangle, \langle r\rangle). & \textit{otherwise} \\ \quad\quad \mid \langle m_p \to m_q, \phi_q\rangle \in \mathcal{E} \wedge g = \mathsf{dec}(\phi_q)\} & \end{cases}$$

In the above formula, p_i (resp., p_o) denotes the height of the operand stack of m at the entry (resp., exit) of m_p. Also, q_i is the height of the stack at the entry of m_q, and TBC^p_m is the decorated bytecode for m_p. We use "_" to indicate that the value at the corresponding position is not relevant.

The function $\mathsf{translate}(m_p, \langle \mathcal{N}, \mathcal{E}\rangle)$ is defined by cases. The first case is applied when m_p is a *sink* node with no out-edges. Otherwise, the second rule introduces an additional procedure m^c_p (c is for *continuation*), which is defined by as many rules as there are out-edges for m_p. These rules capture the different alternatives which execution can follow from m_p. We will unfold calls to m^c_p whenever it is deterministic (m_p has a single out-edge). This results in m_p calling m_q directly.

Example 2. The RBR of the CFG in Fig. 1 consists of the following rules where local variables have the same name as in the source code and o is the *this* object:

$$
\begin{array}{ll}
fact(\langle o, n\rangle, \langle r\rangle) & := \mathsf{init\_local\_vars}(\langle ft, i\rangle),\ fact_1(\langle o, n, ft, i\rangle, \langle r\rangle).\\
fact_1(\langle o, n, ft, i\rangle, \langle r\rangle) & := \mathsf{iconst}(1, s_0),\ \mathsf{istore}(s_0, ft),\ \mathsf{iconst}(1, s_0),\\
 & \quad \mathsf{istore}(s_0, i),\ fact_2(\langle o, n, ft, i\rangle, \langle r\rangle).\\
fact_2(\langle o, n, ft, i\rangle, \langle r\rangle) & := \mathsf{iload}(i, s_0),\ \mathsf{iload}(n, s_1),\\
 & \quad fact_2^c(\langle o, n, ft, i, s_0, s_1\rangle, \langle r\rangle).\\
fact_2^c(\langle o, n, ft, i, s_0, s_1\rangle, \langle r\rangle) & := \mathsf{guard}(\mathsf{icmpgt}(s_0, s_1)),\ fact_4(\langle o, n, ft, i\rangle, \langle r\rangle).\\
fact_2^c(\langle o, n, ft, i, s_0, s_1\rangle, \langle r\rangle) & := \mathsf{guard}(\mathsf{icmple}(s_0, s_1)),\ fact_3(\langle o, n, ft, i\rangle, \langle r\rangle).\\
fact_3(\langle o, n, ft, i\rangle, \langle r\rangle) & := \mathsf{iload}(ft, s_0),\ \mathsf{iload}(i, s_1),\ \mathsf{imul}(s_0, s_1, s_0),\\
 & \quad \mathsf{istore}(s_0, ft),\ \mathsf{iinc}(i, 1),\ fact_2(\langle o, n, ft, i\rangle, \langle r\rangle).\\
fact_4(\langle o, n, ft, i\rangle, \langle r\rangle) & := \mathsf{iload}(ft, s_0),\ \mathsf{ireturn}(s_0, r).
\end{array}
$$

The first rule corresponds to the entry. Block $fact_4$ is a sink block. Blocks $fact_1$ and $fact_3$ have a single out-edge and we have unfolded the continuation. Finally, block $fact_2$ has two out-edges and needs the procedure $fact_2^c$. The RBR from the CFG of *doSum* in Fig. 2 is ($doSum_3$ merges several blocks with one out-edge):

$$
\begin{array}{ll}
doSum(\langle o, x\rangle, \langle r\rangle) & := \mathsf{init\_local\_vars}(\langle\rangle),\ doSum_1(\langle o, x\rangle, \langle r\rangle).\\
doSum_1(\langle o, x\rangle, \langle r\rangle) & := \mathsf{aload}(x, s_0), doSum_1^c(\langle o, x, s_0\rangle, \langle r\rangle).\\
doSum_1^c(\langle o, x, s_0\rangle, \langle r\rangle) & := \mathsf{guard}(\mathsf{nonnull}(s_0)), doSum_3(\langle o, x\rangle, \langle r\rangle).\\
doSum_1^c(\langle o, x, s_0\rangle, \langle r\rangle) & := \mathsf{guard}(\mathsf{null}(s_0)), doSum_2(\langle o, x\rangle, \langle r\rangle).\\
doSum_2(\langle o, x\rangle, \langle r\rangle) & := \mathsf{iconst}(0, s_0), \mathsf{ireturn}(s_0, r).\\
doSum_3(\langle o, x\rangle, \langle r\rangle) & := \mathsf{aload}(o, s_0), \mathsf{aload}(x, s_1), \mathsf{getfield}(List.data, s_1, s_1),\\
 & \quad fact(\langle s_0, s_1\rangle, \langle s_0\rangle), \mathsf{aload}(o, s_1), \mathsf{aload}(x, s_2),\\
 & \quad \mathsf{getfield}(List.next, s_2, s_2), doSum(\langle s_1, s_2\rangle, \langle s_1\rangle),\\
 & \quad \mathsf{iadd}(s_0, s_1, s_0), \mathsf{ireturn}(s_0, r).
\end{array}
$$

We can see that a call to a different method, *fact*, occurs in $doSum_3$. This shows that our RBR allows simultaneously handling the two CFGs in our example. □

Rule-based Programs vs JBC Programs. Given a JBC program P, P_r denotes the RBP obtained from P. Note that, it is trivial to define an *interpreter* (or *abstract machine*) which can execute any P_r and obtain the same return value and termination behaviour as a JVM does for P. RBPs, in spite of their declarative appearance, are in fact imperative programs. As in the JVM, an interpreter for RBPs needs, in addition to a stack for activation records, a global heap. These activation records differ from those in the JVM in that the operand stack is no longer needed (as stack elements are explicit) and in that the scope of variables is no longer associated to methods but rather to rules. In RBPs all rules are treated uniformly, regardless of the method they originate from, so that method borders are somewhat blurred. As in the JVM, call-by-value is used for passing arguments in calls.

3 Proving Termination

This section describes how to prove termination of a JBC program given its RBR. The approach consists of two steps. In the first, we *abstract* the RBR rules by replacing all program data by their corresponding *size*, and replacing calls

corresponding to bytecode instructions by *size constraints* on the values their variables can take. This step results in a Constraint Logic Program (CLP) [16] over integers, where, for any bytecode trace t, there exists a CLP trace t' whose states are abstractions of t states. In particular, every infinite (non terminating) bytecode trace has a corresponding infinite CLP trace, so that termination of the CLP program implies termination of the bytecode program. Note that, unlike in bytecode traces which are always deterministic, the execution of a CLP program can be non-deterministic, due to the precision loss inherent to the abstraction.

In the second step, we apply techniques for proving termination of CLP programs [9], which consist of: (1) analyzing the rules for each method to infer input-output *size relations* between the method input and output variables; (2) using the input-output size relations for the methods in the program, we infer a set of abstract *direct calls-to pairs* which describe, in terms of size-change, all possible calls from one procedure to another; and (3) given this set of abstract direct calls-to pairs, we compute a set of all possible calls-to pairs (direct and indirect), describing all transitions from one procedure to another. Then we focus on the pairs which describe loops, and try to identify *ranking functions* which guarantee the termination of each loop and thus of the original program.

3.1 Abstracting the Rules

As mentioned above, rule abstraction replaces data by the *size* of data, and focuses on relations between data size. For integers, their size is just their integer value [12]. For references, we take their size to be their *path-length* [24], i.e., the length of the maximal path reachable from the reference. Then, bytecode instructions are replaced by constraints on sizes taking into account a Static Single Assignment (SSA) transformation. SSA is needed because variables in CLP programs cannot be assigned more than one value. For example, an instruction $\mathsf{iadd}(s_0, s_1, s_0)$ will be abstracted to $s'_0{=}s_1{+}s_0$ where s'_0 refers to the value of s_0 after executing the instruction. Also, the bytecode $\mathsf{getfield}(f, s_0, s_0)$ is abstracted to $s_0{>}s'_0$ if it can be determined that s_0 (before executing the instruction) does not reference a cyclic data-structure, since the length of the longest-path reachable from s_0 is larger than the length of the longest path reachable from s'_0.

bytecode b	*abstract bytecode* b^α	ρ_{i+1}
$\mathsf{iload}(l_v, s_j)$	$s'_j{=}\rho_i(l_v)$	$\rho_i[s_j{\mapsto}s'_j]$
$\mathsf{iadd}(s_j, s_{j+1}, s_j)$	$s'_j{=}\rho_i(s_j){+}\rho_i(s_{j+1})$	$\rho_i[s_j{\mapsto}s'_j]$
$\mathsf{guard}(icmpgt(s_j, s_{j+1}))$	$\rho_i(s_j){>}\rho_i(s_{j+1})$	ρ_i
$\mathsf{getfield}(f, s_j, s_j)$	if f is of ref. type: $\rho_i(s_j){>}s'_j$ if s_j is not cyclic otherwise $\rho_i(s_j) \geq s'_j$. If f is not of ref. type: *true*	$\rho_i[s_j \mapsto s'_j]$
$\mathsf{putfield}(f, s_j, s_{j+1})$ s.t f is of ref. type	if s_j and s_{j+1} do not share, $s'_k{\leq}\rho_i(s_k){+}\rho_i(s_{j+1})$ for any s_k that shares with s_j, otherwise *true*.	$\rho_i[s_k \mapsto s'_k]$

To implement the SSA transformation we maintain a mapping ρ of variable names (as they appear in the rule) to new variable names (constraint variables). Such a mapping is referred to as a *renaming*. We let $\rho[x \mapsto y]$ denote the

modification of the renaming ρ such that it maps x to the new variable y. We denote by $\rho[\bar{x} \mapsto \bar{y}]$ the mapping of each element in $\bar{x}$ to a corresponding one in $\bar{y}$.

Definition 5 (abstract compilation). *Let* $\mathrm{R} \equiv p(\overline{x}, \overline{y}) := b_1, \ldots, b_n$ *be a rule. Let* ρ_i *be a renaming associated with the point before each* b_i *and let* ρ_1 *be the identity renaming (on the variables in the rule). The abstraction of* R *is denoted* R^α *and takes the form* $p(\overline{x}, \overline{y}') := b_1^\alpha, \ldots, b_n^\alpha$ *where* b_i^α *are computed iteratively from left to right as follows:*

1. *if* b_i *is a bytecode instruction or a guard, then* b_i^α *and* ρ_{i+1} *are obtained from a predefined lookup table similar to the one above.*
2. *if* b_i *is a call to a procedure* $q(\overline{w}, \overline{z})$*, then the abstraction* b_i^α *is* $q(\overline{w}', \overline{z}')$ *where each* $w'_k \in \overline{w}'$ *is* $\rho_i(w_k)$*, variables* $\overline{z}'$ *are fresh, and* $\rho_{i+1} = \rho_i[\overline{z} \mapsto \overline{z}', \overline{u} \mapsto \overline{u}']$ *where* $\overline{u}'$ *are also fresh variables and* $\overline{u}$ *is the set of all variables in* $\overline{w}$ *which reference data-structures that can be modified when executing* q *and those that share (i.e., might have common regions in the heap) with them.*
3. *at the end we define each* $y'_i \in \overline{y}'$ *to be the constrained variable* $\rho_{n+1}(y_i)$*.*

In addition, all reference *variables are (implicitly) assumed to be non-negative.*

Note that in point 2 above, the set of variables such that the data-structures they point to may be modified during the execution of q can be approximated by applying constancy analysis [14], which aims at detecting the method arguments that remain constant during execution, and sharing analysis [23] which aims at detecting reference variables that might have common regions on the heap. Also, the non-cyclicity condition required for the abstraction of getfield can be verified by non-cyclicity analysis [22]. In what follows, for simplicity, we assume that abstract rules are normalized to the form $p(\overline{x}, \overline{y}) := \varphi, p_1(\overline{x}_1, \overline{y}_1), \ldots, p_j(\overline{x}_j, \overline{y}_j)$ where φ is the conjunction of the (linear) size constraints introduced in the abstraction and each $p_i(\overline{x}_i, \overline{y}_i)$ is a call to a procedure (i.e., block or method).

Example 3. Recall the following rule from Ex. 2 (on the left) and its abstraction (on the right) where the renamings are indicated as comments.

$fact_3(\langle o, n, ft, i\rangle, \langle r\rangle) :=$	$fact_3(\langle o, n, ft, i\rangle, \langle r'\rangle) :=$	$\%\ \rho_1 = id$
$\mathsf{iload}(ft, s_0),$	$s'_0 = ft,$	$\%\ \rho_2 = \rho_1[s_0 \mapsto s'_0]$
$\mathsf{iload}(i, s_1),$	$s'_1 = i,$	$\%\ \rho_3 = \rho_2[s_1 \mapsto s'_1]$
$\mathsf{imul}(s_0, s_1, s_0),$	$true,$	$\%\ \rho_4 = \rho_3[s_0 \mapsto s''_0]$
$\mathsf{istore}(s_0, ft),$	$ft' = s''_0,$	$\%\ \rho_5 = \rho_4[ft \mapsto ft']$
$\mathsf{iinc}(i, 1),$	$i' = i + 1,$	$\%\ \rho_6 = \rho_5[i \mapsto i']$
$fact_2(\langle o, n, ft, i\rangle, \langle r\rangle).$	$fact_2(\langle o, n, ft', i'\rangle, \langle r'\rangle).$	$\%\ \rho_7 = \rho_6[r \mapsto r']$

Note that imul is abstracted to *true*, since it imposes a non-linear constraint. □

3.2 Input Output Size-Relations

We consider the abstract rules obtained in the previous step to infer an abstraction (w.r.t. size) of the input-output relation of the program blocks. Concretely,

we infer *input-output size relations* of the form $p(\overline{x},\overline{y}) \leftarrow \varphi$, where φ is a constraint describing the relation between the sizes of the input $\overline{x}$ and the output $\overline{y}$ upon exit from p. This information is needed since output of one call may be input to another call. E.g., consider the following contrived abstract rule $p(\langle x\rangle,\langle r\rangle)$:=$\{x>0,x>z\}, q(\langle z\rangle,\langle y\rangle), p(\langle y\rangle,\langle r\rangle)$. To prove termination, it is crucial to know the relation between x in the head and y in the recursive call to p. This requires knowledge about the input-output size relations for $q(\langle z\rangle,\langle y\rangle)$. Assuming this to be $q(\langle z\rangle,\langle y\rangle) \leftarrow z>y$, we can infer $x>y$. Since abstract programs are CLP programs, inferring relations can rely on standard techniques [4].

Computing an approximation of input-output size relation requires a global fixpoint. In practice, we can often take a trivial over-approximation where for all rules there is no information, namely, $p(\overline{x},\overline{y}) \leftarrow \mathit{true}$. This can prove termination of many programs, and results in a more efficient implementation. It is not enough in cases as the above abstract rule, which however, in our experience, often does not occur in imperative programs.

3.3 Call-to Pairs

Consider again the abstract rule from Ex. 3 which (ignoring the output variable) is of the form $\mathit{fact}_3(\bar{x}) := \varphi, \mathit{fact}_2(\bar{z})$. It means that whenever execution reaches a call to $\mathit{fact}_3(\bar{x})$ there will be a subsequent call to $\mathit{fact}_2(\bar{z})$ and the constraint φ holds. In general, subsequent calls may arise also from rules which are not binary. Given an abstract rule of the form $p_0 := \varphi, p_1, \ldots, p_n$, a call to p_0 may lead to a call to p_i, $1{\leq}i{\leq}n$. Given the input-output size relations for the individual calls $p_1,\ldots,p_{i-1}$, we can characterize the constraint for a transition between the subsequent calls p_0 and p_i by adding these relations to φ. We denote a pair of such subsequent calls by $\langle p_0(\overline{x}) \rightsquigarrow p_i(\overline{y}), \varphi_i\rangle$ and call it a *calls-to pair*.

Definition 6 (direct calls-to pairs). *Given a set of abstract rules $\mathcal{A}$ and its input-output size relations $I_{\mathcal{A}}$, the* direct calls-to pairs *induced by $\mathcal{A}$ and $I_{\mathcal{A}}$ are:*

$$C_{\mathcal{A}} = \left\{ \langle p(\overline{x}) \rightsquigarrow p_i(\overline{x}_i), \psi\rangle \;\middle|\; \begin{array}{l} p(\overline{x},\overline{y}) := \varphi, p_1(\overline{x}_1,\overline{y}_1),\ldots,p_j(\overline{x}_j,\overline{y}_j){\in}\mathcal{A}, \\ i{\in}\{1,\ldots,j\},\ \forall 0{<}k{<}i.\ p_k(\overline{x}_k,\overline{y}_k) \leftarrow \varphi_k{\in}I_{\mathcal{A}} \\ \psi = \bar{\exists}\overline{x} \cup \overline{x}_i.\varphi \wedge \varphi_1 \wedge \ldots \wedge \varphi_{i-1} \end{array} \right\}$$

where $\bar{\exists}v$ means eliminating all variables but v from the corresponding constraint.

Example 4. Consider the rule for *doSum* in Ex. 2: note that input-output relations for *fact* and *doSum* are *true*. Direct calls-to pairs for those rules are:

$$\begin{array}{l} \langle \mathit{doSum}(o,x) \rightsquigarrow \mathit{doSum}_1(o',x'), \{x'{=}x, o'{=}o\}\rangle \\ \langle \mathit{doSum}_1(o,x) \rightsquigarrow \mathit{doSum}_1^c(o',x',s_0), \{x'{=}x, o'{=}o, s_0{=}x\}\rangle \\ \langle \mathit{doSum}_1^c(o,x,s_0) \rightsquigarrow \mathit{doSum}_3(o',x'), \{x'{=}x, o'{=}o, s_0{>}0\}\rangle \\ \langle \mathit{doSum}_1^c(o,x.s_0) \rightsquigarrow \mathit{doSum}_2(o',x'), \{x'{=}x, o'{=}o, s_0{=}0\}\rangle \\ \langle \mathit{doSum}_3(o,x) \rightsquigarrow \mathit{fact}(s_0',s_1''), \{s_0'{=}o\}\rangle \\ \langle \mathit{doSum}_3(o,x) \rightsquigarrow \mathit{doSum}(s_1''',s_2''), \{s_1'''{=}o, x{>}s_2''\}\rangle \end{array}$$

In the last rule, s_2'' corresponds to $x.\mathit{next}$, so that we have the constraint $x{>}s_2''$. It can be seen that since the list is not cyclic and does not share with other

variables, size analysis finds the above decreasing of its size $x{>}s_2''$. Note also that all variables corresponding to references are assumed to be non-negative. Similarly, we can obtain direct calls-to pairs for the rule of *fact*. □

It should be clear that the set of direct calls-to pairs relations $C_{\mathcal{A}}$ is also a binary CLP program that we can execute from a given goal. A key feature of this binary program is that if an infinite trace can be generated using the abstract program described in Sec. 3.1, then an infinite trace can be generated using this binary CLP program [10]. Therefore, absence of such infinite traces (i.e., termination) in the binary program $C_{\mathcal{A}}$ implies absence of infinite traces in the abstract bytecode program, as well as in the original bytecode program.

Theorem 1 (Soundness). *Let P be a JBC program and $C_{\mathcal{A}}$ the set of direct calls-to pairs computed from P. If there exists a non-terminating trace in P then there exists a non-terminating derivation in $C_{\mathcal{A}}$.*

Intuitively, the result follows from the following points. By construction, the RBP captures all possibly non-terminating traces in the original program. By the correctness of size analysis, we have that, given a trace in the RBP, there exists an equivalent one in $C_{\mathcal{A}}$, among possibly others. Therefore, termination in $C_{\mathcal{A}}$ entails termination in the JBC program.

3.4 Proving Termination of the Binary Program $C_{\mathcal{A}}$

Several automatic termination tools and methods for proving termination of such binary constraint programs exists [9,10,17]. They are based on the idea of first computing all possible calls-to pair from the direct ones, and then finding a ranking function for each recursive calls-to pairs, which is sufficient for proving termination. Computing all possible calls-to pairs, usually called the *transitive closure* $C_{\mathcal{A}}^*$, can be done by starting from the set of direct calls-to pairs $C_{\mathcal{A}}$, and iteratively adding to it all possible compositions of its elements until a fixed-point is reached. Composing two calls-to pairs $\langle p(\overline{x}) \rightsquigarrow q(\overline{y}), \varphi_1\rangle$ and $\langle q(\overline{w}) \rightsquigarrow r(\overline{z}), \varphi_2\rangle$ returns the new calls-to pair $\langle p(\overline{x}) \rightsquigarrow r(\overline{z}), \bar{\exists}\bar{x} \cup \bar{z}.\varphi_1 \wedge \varphi_2 \wedge (\bar{y} = \bar{w})\rangle$.

Example 5. Applying the transitive closure on the direct calls-to pairs of Ex. 4, we obtain, among many others, the following calls-to pairs. Note that x (resp. i) strictly decreases (resp. increases) at each iteration of its corresponding loop:

$$\langle doSum(o,x) \rightsquigarrow doSum(o',x'), \{o'{=}o, x{>}x', x{\geq}0\}\rangle$$
$$\langle fact_2(o,n,ft,i) \rightsquigarrow fact_2(o',n',ft',i'), \{o'{=}o, n'{=}n, i'{>}i, i{\geq}1, n{\geq}i'{-}1\}\rangle \quad \square$$

As already mentioned, in order to prove termination, we focus on *loops* in $C_{\mathcal{A}}^*$. Loops are the *recursive* entities of the form $\langle p(\overline{x}) \rightsquigarrow p(\overline{y}), \varphi\rangle$ which indicate that a call to a program point p with values $\bar{x}$ eventually leads to a call to the same program point with values $\bar{y}$ and that φ holds between $\overline{x}$ and $\overline{y}$. For each loop, we seek a *ranking function* F over a well-founded domain such that $\varphi \models \mathrm{F}(\overline{x}){>}\mathrm{F}(\overline{y})$. As shown in [9,10], finding a ranking function for every recursive calls-to pair implies termination. Computing such functions can be done, for instance, as described in [21]. As an example, for the loops in Ex. 5 we get the following ranking functions: $\mathrm{F}_1(o,x){=}x$ and $\mathrm{F}_2(o,n,ft,i){=}n{-}i{+}1$.

3.5 Improving Termination Analysis by Extracting Nested Loops

In proving termination of JBC programs, one important question is whether we can prove termination at the JBC level for a class of programs which is comparable to the class of Java *source* programs for which termination can be proved using similar technology. As can be seen in Sec. 4, directly obtaining the RBR of a bytecode program is non-optimal, in the sense that proving termination on it may be more complicated than on the source program. This happens because, while in source code it is easy to reason about a nested loop independently of the outer loop, loops are not directly visible when control flow is unstructured. Loop extraction is useful for our purposes since nested loops can be dealt with one at a time. As a result, finding a ranking function is easier, and computing the closure can be done locally in the strongly connected components. This can be crucial in proving the termination of programs with nested loops.

To improve the accuracy of our analysis, we include a component which can detect and extract loops from CFGs. Due to space limitations, we do not describe how to perform this step here (more details in work about decompilation [2], where loop extraction has received considerable attention). Very briefly, when a loop is extracted, a new CFG is created. As a result, a method can be converted into several CFGs. These ideas fit very nicely within our RBR, since calls to loops are handled much in the same way as calls to other methods.

4 Experimental Results

Our prototype implementation is based on the size analysis component of [1] and extends it with the additional components needed to prove termination. The analyzer can also output the set of direct call-pairs, which allows using existing termination analyzers based on similar ideas [10,17]. The system is implemented in Ciao Prolog, and uses the Parma Polyhedra Library (PPL) [3].

Table 1 shows the execution times of the different steps involved in proving the termination of JBC programs, computed as the arithmetic mean of five runs. Experiments have been performed on an Intel 1.86 GHz Pentium M with 1 GB of RAM, running Linux on a 2.6.17 kernel. The table shows a range of benchmarks for which our system can prove termination, and which are meant to illustrate different features. We show classical recursive programs such as *Hanoi*, *Fibonacci*, *MergeList* and *Power*. Iterative programs *DivByTwo* and *Concat* contain a single loop, while *Sum*, *MatMult* and *BubbleSort* are implemented with nested loops. We also include programs written in object-oriented style, like *Polynomial*, *Incr*, *Scoreboard*, and *Delete*. The remaining benchmarks use data structures: arrays (*ArrayReverse*, *MatMultVector*, and *Search*); linked lists (*Delete* and *ListReverse*); and binary trees (*BST*).

Columns **CFG**, **RBR**, **Size**, **TC**, **RF**, **Total**$_1$ contain the running times (in ms) required for the CFG (including loop extraction), the RBR, the size analysis (including input-output relations), the transitive closure, the ranking functions and the total time, respectively. Times are high, as the implementation has been developed to check if our approach is feasible, but is still preliminary. The most

Table 1. Measured time (in ms) of the different phases of proving termination

Benchmark	CFG	RBR	Size	TC	RF	$\mathbf{Total}_1$	Termin	$\mathbf{Total}_2$	Ratio
Polynomial	138	12	260	1453	26	1890	yes	2111	1.12
DivByTwo	52	4	168	234	4	462	yes	538	1.17
EvenDigits	59	7	383	1565	17	2030	yes	2210	1.09
Factorial	43	3	46	268	3	363	yes	353	0.97
ArrayReverse	58	5	208	339	24	635	yes	834	1.32
Concat	65	8	660	943	38	1715	yes	3815	2.23
Incr	35	12	854	4723	28	5652	yes	6590	1.17
ListReverse	21	5	141	310	5	481	yes	515	1.07
MergeList	107	23	130	5184	21	5464	yes	5505	1.01
Power	14	3	72	357	9	454	yes	459	1.01
Cons	25	7	65	1318	10	1424	yes	1494	1.05
ListInter	136	22	585	9769	49	10560	yes	27968	2.65
SelectOrd	154	16	1298	4076	48	5592	no	25721	4.60
DoSum	57	10	64	923	6	1060	yes	1069	1.01
Delete	121	14	54	2418	1	2608	yes	33662	12.91
MatMult	240	11	2411	4646	294	7602	no	32212	4.24
MatMultVector	254	15	2563	8744	242	11817	no	34688	2.94
Hanoi	39	5	172	979	3	1198	no	1198	1.00
Fibonacci	23	3	90	290	5	411	yes	401	0.98
BST	68	12	97	4643	18	4838	yes	4901	1.01
BubbleSort	152	12	1125	4366	83	5738	no	14526	2.53
Search	65	11	307	756	11	1150	yes	1430	1.24
Sum	64	7	480	1758	35	2343	no	5610	2.39
FactSumList	65	12	80	961	5	1123	yes	1306	1.16
Scoreboard	268	23	1597	4393	81	6362	no	32999	5.19

expensive steps are the size analysis and the transitive closure, since they require global analysis. Last three columns show the benefits of loop extraction. **Termin** tells if termination can be proven (using polyhedra) without extraction. In seven cases, termination is only proven if loop extraction is performed. $\mathbf{Total}_2$ shows the total time required to check termination without loop extraction. **Ratio** compares $\mathbf{Total}_2$ with $\mathbf{Total}_1$ ($\mathbf{Total}_2/\mathbf{Total}_1$), showing that, in addition to improving precision, loop extraction is beneficial for efficiency, since **Ratio** ≥ 1 in most cases, and can be as high as 12.91 in *Delete*. Note that termination of these programs may be proved without loop extraction by using other domains such as *monotonicity constraints* [7]. However, we argue that loop extraction is beneficial as it facilitates reasoning on the loops separately. Also, if it fails to prove termination, it reports the possibly non-terminating loops.

5 Conclusions and Related Work

We have presented a termination analysis for (sequential) JBC which is, to the best of our knowledge, the first approach in this direction. This analysis

successfully deals with the challenges related to the low-level nature of JBC, and adapts standard techniques used, in other settings, in decompilation and termination analysis. Also, we believe that many of the ideas presented in this paper are also applicable to termination analysis of low-level languages in general, and not only JBC. We have used the notion of path-length to measure the size of data structures on the heap. However, our approach is parametric on the abstract domain used to measure the size. As future work, we plan to implement non-cyclicity analysis [22], constancy analysis [14], and sharing analysis [23], and to enrich the transitive closure components with monotonicity constraints [7]. Unlike polyhedra, monotonicity constraints can handle disjunctive information which is often crucial for proving termination. In [5], a termination analysis for C programs, based on binary relations similar to ours, is proposed. It uses separation logic to approximate the heap structure, which in turn allows handling termination of programs manipulating cyclic data structures. We believe that, for programs whose termination does not depend on cyclic data-structures, both approaches deal with the same class of programs. However, ours might be more efficient, as it is based on a simpler abstract domains (a detailed comparison is planned for future work). Recently, a novel termination approach has been suggested [8]. It is based on cyclic proofs and separation logic, and can even handle complicated examples as the reversal of panhandle data-structures. It is not clear to us how practical this approach is.

Acknowledgments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. Samir Genaim was supported by a *Juan de la Cierva* Fellowship awarded by the Spanish Ministry of Science and Education. Part of this work was performed during a research stay of Michael Codish at UPM supported by a grant from the Secretaría de Estado de Educación y Universidades, Spanish Ministry of Science and Education. The authors would like to thank the anonymous referees for their useful comments.

References

1. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of Java Bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, Springer, Heidelberg (2007)
2. Allen, F.: Control flow analysis. In: Symp. on Compiler optimization (1970)
3. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.: Possibly not closed convex polyhedra and the Parma Polyhedra Library. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, Springer, Heidelberg (2002)
4. Benoy, F., King, A.: Inferring Argument Size Relationships with CLP(R). In: Gallagher, J.P. (ed.) LOPSTR 1996. LNCS, vol. 1207, pp. 204–223. Springer, Heidelberg (1997)

5. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.: Automatic termination proofs for programs with shape-shifting heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
6. Bradley, A., Manna, Z., Sipma, H.: Termination of polynomial programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, Springer, Heidelberg (2005)
7. Brodsky, A., Sagiv, Y.: Inference of Inequality Constraints in Logic Programs. In: Proceedings of PODS 1991, pp. 95–112. ACM Press, New York (1991)
8. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: Proceedings of POPL-35 (January 2008)
9. Bruynooghe, M., Codish, M., Gallagher, J., Genaim, S., Vanhoof, W.: Termination analysis of logic programs through combination of type-based norms. ACM TOPLAS 29(2) (2007)
10. Codish, M., Taboch, C.: A semantic basis for the termination analysis of logic programs. J. Log. Program. 41(1), 103–123 (1999)
11. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI (2006)
12. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. POPL. ACM Press, New York (1978)
13. DeLine, R., Leino, R.: BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft (2005)
14. Genaim, S., Spoto, F.: Technical report, Personal Communication (2007)
15. Giesl, J., Schneider-Kamp, P., Thiemann, R.: AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130. Springer, Heidelberg (2006)
16. Jaffar, J., Maher, M.: Constraint Logic Programming: A Survey. Journal of Logic Programming 19(20), 503–581 (1994)
17. Lee, C., Jones, N., Ben-Amram, A.: The size-change principle for program termination. In: Proc. POPL. ACM Press, New York (2001)
18. Lindenstrauss, N., Sagiv, Y.: Automatic termination analysis of logic programs. In: ICLP (1997)
19. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. A-W (1996)
20. Necula, G.: Proof-Carrying Code. In: POPL 1997. ACM Press, New York (1997)
21. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937. Springer, Heidelberg (2004)
22. Rossignoli, S., Spoto, F.: Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855. Springer, Heidelberg (2005)
23. Secci, S., Spoto, F.: Pair-sharing analysis of object-oriented programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 320–335. Springer, Heidelberg (2005)
24. Spoto, F., Hill, P.M., Payet, E.: Path-length analysis for object-oriented programs. In: Proc. EAAI (2006)
25. Spoto, F., Jensen, T.: Class analyses as abstract interpretations of trace semantics. ACM Trans. Program. Lang. Syst. 25(5), 578–630 (2003)
26. Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: Proc. of CASCON 1999, pp. 125–135 (1999)

Sessions and Pipelines for Structured Service Programming*

Michele Boreale[1], Roberto Bruni[2], Rocco De Nicola[1], and Michele Loreti[1]

[1] Dipartimento di Sistemi e Informatica, Università di Firenze
{boreale,denicola,loreti}@dsi.unifi.it
[2] Dipartimento di Informatica, Università di Pisa
bruni@di.unipi.it

Abstract. Service-oriented computing is calling for novel computational models and languages with primitives for client-server interaction, orchestration and unexpected events handling. We present CaSPiS, a process calculus where the notions of session and pipelining play a central role. Sessions are two-sided and can be equipped with protocols executed by each side. Pipelining permits orchestrating the flow of data produced by different sessions. The calculus is also equipped with operators for handling (unexpected) termination of the partner's side of a session. Several examples are presented to provide evidence for the flexibility of the chosen set of primitives. Our main result shows that in CaSPiS it is possible to program a "graceful termination" of nested sessions, which guarantees that no session is forced to hang forever after the loss of its partner.

1 Introduction

The explosive growth of the Web has led to the widespread use of *de facto* standards for naming schemes (URI, URL), communication protocols (SOAP, HTTP, TCP/IP) and message format (XML). These three components have been used as the basis for building communication centered applications distributed over the web, often referred to as web services, and have put many expectations of the IT community on the growth of a new computational paradigm known as *Service-Oriented Computing* (SOC).

In SOC a main issue is the scalability of the proposed languages, models and techniques. Our belief is that the amount of complexity originated in the so-called global computing applications can be handled by considering well-structured and tightly disciplined approaches to the modeling of interaction. Well studied process algebras, like for instance π-calculus [21], have been used as a foundational model for SOC. However, we see two main problems along this way. The first is separation of concerns: SOC has different first-class aspects that would be mixed up and obfuscated when encoded via π-calculus channels. The second is that π-calculus communication primitives seem too liberal: the lack of structure in the communication topology increases the complexity of the analysis.

Here, we try to center the design of a new process calculus around a few prominent aspects of SOC applications, possibly reusing elegant patterns appeared in different

* Research supported by the Project FET-GC II IST-2005-16004 SENSORIA and by the Italian FIRB Project TOCAI.IT.

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 19–38, 2008.

proposals. The three aspects that motivated our design choices are service autonomy, client-service interaction, and orchestration. Each aspect is briefly discussed below.

Services are heterogeneous computational entities, that are developed separately and often are scarcely reliable. Each service has *full autonomy* in denying a request or abandoning a pending interaction. A language for SOC should fix some standard mechanism for programming such decisions and to handle their consequences.

A language for SOC should support programming of complex and safe client-service interactions. By *interaction*, we mean the main unit of activity in a service-oriented application, which is essentially a conversation between a client and an instance of a service. The interaction may be *complex* as it will in general comprise both the exchange of several messages and the invocation of subsidiary services. As an example, consider a travel agent service that offers packages for organized trips. We expect that a dialogue takes place between the customer and the service, to let the service learn the customer preferences, let the customer select one among available packages, confirm or cancel the choice, and so on. In the course of this interaction, the service may need to invoke third party services to get, say, up-to-date flight or hotel information. By *safe*, we mean that, in principle, the involved parties should always be able either to complete the interaction or to recover from errors that prevent its completion, like, in the scenario above, one of the third-party services unexpectedly abandoning the conversation.

Orchestration is the process of assembling different services to build a new one or simply to perform a specific task. A central aspect of orchestration is the organization of the data flow among different activities. This flow also determines synchronization of activities. For instance, in the scenario outlined above, upon client's request, the travel agent service can start two new concurrent activities to get hotel and flight information (by invoking two subsidiary services), wait for their results and finally pass them on to the customer.

Motivated by these considerations, we introduce a language where *sessions* and *pipelines* are viewed as natural tools for monitoring the communication graph by structuring client-service interaction and orchestration, respectively. Autonomy is reflected as the ability to leave sessions. We name the new calculus *CaSPiS (Calculus of Sessions and Pipelines)*.

The use of session is a more abstract, alternative solution w.r.t. the W3C proposal of correlation sets. Here we use a name-scoping mechanism à la π-calculus to handle sessions. Pipelines have been inspired by Cook and Misra's Orc [22], a basic and elegant programming model for structured orchestration of services. In this light, they are seen as a convenient mechanism for modeling the flow of data between local processes: it is more general than sequential composition, better suited w.r.t. concurrency and does not require the explicit and improper use of channels for orchestration tasks. CaSPiS evolved from SCC *(Serviced Centered Calculus)* [3], a calculus that arose from a coordinated effort within the EU funded project SENSORIA [23]. The improvements and relationship of our work w.r.t. other proposals is discussed in the concluding section. In the rest of this section, we incrementally describe and motivate the main features of CaSPiS and discuss our results.

CaSPiS in a nutshell. In CaSPiS, service definitions and invocations are written like input and output prefixes in CCS. Thus `sign`.P defines a service `sign` that can be

invoked by $\overline{\texttt{sign}}.Q$. There is an important difference, though, as the bodies P and Q are not quite continuations, but rather protocols that, within a session, govern interaction between (instances of) the client and the server. As an example:

$$!\texttt{sign}.(?x)(\nu t)\langle\{x,t\}_k\rangle \qquad \text{and} \qquad \overline{\texttt{sign}}.\langle \mathsf{plan}\rangle(?y)\langle y\rangle^{\uparrow}$$

are respectively: a (replicated and thus persistent) service whose instance waits for a digital document x, generates a fresh nonce t and then sends back both the document and the nonce signed with a key k; and a client that passes the argument plan to the service, then waits for the signed response from the server and returns this value outside the session as a result.

Synchronization of $s.P$ and $\overline{s}.Q$ leads to the creation of a new session, identified by a fresh name r that can be viewed as a private, synchronous channel binding caller and callee. Since client and service may be far apart, a session naturally comes with two sides, written $r \rhd P$ and $r \rhd Q$, with r bound somewhere above them by (νr). Values produced by P can be consumed by Q, and vice-versa: this permits description of interaction patterns more complex than the usual *one-way* and *request-response*. Rules governing creation and scoping of sessions are based on those of the restriction operator in the π-calculus. Note that multiple invocations to the same persistent service will yield separate sessions and that hierarchies of nested sessions, like $r_1 \rhd (r_2 \rhd P_2 | r_3 \rhd P_3)$ can arise if services are invoked within a running session. In the above case of service `sign` the triggered session is

$$!\texttt{sign}.(?x)(\nu t)\langle\{x,t\}_k\rangle \quad | \quad (\nu r)\big(r \rhd (?x)(\nu t)\langle\{x,t\}_k\rangle \quad | \quad r \rhd \langle \mathsf{plan}\rangle(?y)\langle y\rangle^{\uparrow}\big).$$

Here, after one reduction step where the value $\langle \mathsf{plan}\rangle$ available on the client-side is transmitted to the pending request $(?x)$ on the service-side (with x substituted by plan in the continuation), we get:

$$!\texttt{sign}.(?x)(\nu t)\langle\{x,t\}_k\rangle \quad | \quad (\nu r,t)\big(r \rhd \langle\{\mathsf{plan},t\}_k\rangle \quad | \quad r \rhd (?y)\langle y\rangle^{\uparrow}\big).$$

Then, the digitally signed value $\{\mathsf{plan},t\}_k$ is computed at the service-side (for some fresh nonce t) and sent back to the client-side:

$$!\texttt{sign}.(?x)(\nu t)\langle\{x,t\}_k\rangle \quad | \quad (\nu r,t)\big(r \rhd \mathbf{0} \quad | \quad r \rhd \langle\{\mathsf{plan},t\}_k\rangle^{\uparrow}\big).$$

The remaining activity will be then performed by the client protocol: $r \rhd \langle\{\mathsf{plan},t\}_k\rangle^{\uparrow}$ will emit $\{\mathsf{plan},t\}_k$ outside the (client-side of the) session, becoming the inert process $r \rhd \mathbf{0}$ (as already happened to the service side). In fact, values can be returned outside a session to the enclosing environment using the return operator, $\langle\cdot\rangle^{\uparrow}$.

Return values can be consumed by other sessions, or used to invoke other services, to start new activities. This is achieved using the pipeline operator $P > Q$. Here, a new instance of process Q is activated each time P emits a value that Q can consume. Notably, the new instance will run within the same session as P, not in a fresh one. For instance, what follows is a client that invokes the service `sign` twice and then stores the obtained signed documents by invoking a suitable service `store`:

$$\big(\overline{\texttt{sign}}.\langle \mathsf{plan}_1\rangle(?y)\langle y\rangle^{\uparrow} \quad | \quad \overline{\texttt{sign}}.\langle \mathsf{plan}_2\rangle(?y)\langle y\rangle^{\uparrow}\big) \quad > \quad (?z)\overline{\texttt{store}}.\langle z\rangle\,.$$

The above description collects the main features of what we call the close-free fragment of CaSPiS that also includes guarded sums and input prefixes with pattern matching.

The distinguishing feature of our calculus is the presence of novel primitives to explicitly program session termination, to handle (unexpected or programmed) session termination and to garbage-collect terminated sessions. As explained before, session units must be able to autonomously decide to abandon the session they are running in. But since sessions units model client-server interactions, their termination must be programmed carefully, especially in the presence of nesting. The command close is used to terminate the enclosing session side. A terminated session enters the special state $\blacktriangleright P$ that recursively terminates any other session side nested in P. Note that the execution of a close can depend on some local choice as well as be guarded by the input of some data from the opposite session side.

The idea is that upon termination of a session side, the opposite session side will be informed and take some proper counteraction if needed. To achieve this, upon creation of a session, one associates with the fresh session r a pair of names (k_1, k_2), identifying a pair of *termination handlers*, one for each side. Then, right after execution of close a signal $\dagger(k_i)$ is sent to the termination-handler service k_i listening at the *opposite* side of the session. This handler will manage the appropriate actions. Since the name k_i must be known to the current side of the session, the more general syntax for sessions is $r \rhd_k P$ where the subscript k refers to the termination handler of the opposite side. To sum up the above discussion:

$$r \rhd_k \left(\mathsf{close} \mid P\right) \qquad \text{may evolve to} \qquad \dagger(k) \mid \blacktriangleright P\,.$$

Information about which termination handlers must be used is established at invocation time. To this purpose, the more general syntax for invocation is $\overline{s}_{k_1}.Q$. It mentions a name k_1 at which the termination handler of the client-side is listening. Symmetrically, the more general syntax for service definition is $s_{k_2}.P$, which mentions a name k_2 at which the termination handler of the service-side is listening. Then

$$\overline{s}_{k_1}.Q \mid s_{k_2}.P \qquad \text{can evolve to} \qquad (\nu r)(r \rhd_{k_2} Q \mid r \rhd_{k_1} P).$$

This way, if Q terminates with close, the termination handler k_2 of the callee will be activated, and vice versa, if P terminates then k_1 will be activated.

The mechanism of termination handlers is very expressive and flexible. Even if it may look overcomplicated to use, we emphasize that, up to our knowledge, this is the only proposal able to guarantee a disciplined termination of nested sessions. We conjecture that any mechanism of this kind would be very complicated to handle in say π-calculus.

Structure of the paper. For the sake of presentation, we introduce CaSPiS in two steps. First, we present the fragment without session-closing primitives, along with its labelled transition system semantics and well-formedness criteria (Section 2), with several nice examples in Section 3. Then we consider the full version of the calculus with session-closing primitives (Section 4). Our main result shows that, with the given primitives, it is possible to program what we call "graceful termination" of sessions (Section 5).

Specifically, we define a notion of *balanced* process (where all session-sides are pairwise balanced) and prove that any unbalanced state reachable from a balanced one can still become balanced in a finite number of steps. Final remarks, related work and future research avenues are exposed in Section 6.

2 The close-Free Fragment of CaSPiS

In this section, we introduce the fragment of CaSPiS without the constructs for handling session termination. The full calculus will be formalized only after the reader has gained some familiarity with the base constructs.

2.1 Syntax

Let $\mathcal{N}_{\text{srv}}$ and $\mathcal{N}_{\text{sess}}$ be two disjoint countable sets, respectively of *service* names $s, s', \ldots$ and of *session* names $r, r' \ldots$. We assume a countable set of *names* $\mathcal{N}$ ranged over by $n, n', \ldots$ that contains $\mathcal{N}_{\text{srv}} \cup \mathcal{N}_{\text{sess}}$ and such that $\mathcal{N} \setminus (\mathcal{N}_{\text{srv}} \cup \mathcal{N}_{\text{sess}})$ is infinite, and let $x, y, \ldots, u, v \ldots$ range over $\mathcal{N} \setminus \mathcal{N}_{\text{sess}}$. We also assume a signature Σ of *constructors* $f, f', \ldots$, each coming with a fixed arity, such that Σ is disjoint from $\mathcal{N}$. We shall use $\tilde{\cdot}$ to denote sequences of items. The syntax of the basic fragment of CaSPiS is reported in Figure 1, where operators are listed in decreasing order of precedence.

To improve usability, structured values V can be built via Σ, and selection patterns F can be used to guard choices (via pattern matching). For simplicity, we consider as basic values only value expressions V built out of constructors in Σ and names $x, y, \ldots, u, v, \ldots$, the latter playing the role of variables or basic values depending on the context. We leave the signature Σ unspecified, but in several examples we shall assume Σ contains tuple constructors $\langle \cdot, \ldots, \cdot \rangle$ of arbitrary arity. Richer languages of expressions, comprising specific data values and evaluation mechanisms, are easy to accommodate. Finally, it is worth to note that session names $r, r', \ldots$ do *not* appear in values or patterns: this implies that they cannot be passed around, as it will be evident from the operational semantics (see Section 2.2).

As expected, in $(\nu n)P$, the restriction (νn) binds free occurrences of n in P, while in $(F)P$ an $?x$ in the pattern F binds the free occurrences of name x in P. We denote by $\text{bn}(F)$ the set of names x such that $?x$ occurs in F. Processes are identified up to alpha-equivalence. The guarded sum with $I = \emptyset$ will also be denoted by $\mathbf{0}$. Trailing $\mathbf{0}$'s will often be omitted.

We will let $\sigma, \sigma', \sigma_1, \ldots$ range over substitutions, that is, finite partial functions from $\mathcal{N}$ to $\mathcal{N}$. In particular, we let $[u/x]$ denote the substitution that maps x to u. For any term T, we let $T\sigma$ denote the result of the capture-avoiding substitution of the free occurrences of x by $\sigma(x)$, for each $x \in \text{dom}(\sigma)$.

Let us anticipate that the process grammar defined in Figure 1 should be pragmatically considered as a run-time syntax. In particular, sessions $r \triangleright P$ can be generated at run-time, upon service invocation, but a programmer is not expected to explicitly use them. These considerations will lead us to impose some constraints on the somewhat too liberal syntax of Figure 1 and to introduce a notion of well-formedness (see Section 2.3).

$$
\begin{array}{lll}
P,Q ::= \sum_{i\in I}\pi_i P_i & \text{Guarded Sum} & \pi ::= (F) \quad \text{Abstraction} \\
\mid s.P & \text{Service Definition} & \mid \langle V\rangle \quad \text{Concretion} \\
\mid \overline{s}.P & \text{Service Invocation} & \mid \langle V\rangle^{\uparrow} \quad \text{Return} \\
\mid r \rhd P & \text{Session} & \\
\mid P > Q & \text{Pipeline} & V ::= u \mid f(\tilde{V}) \quad \text{Value } (f\in\Sigma) \\
\mid P|Q & \text{Parallel Composition} & \\
\mid (\nu n)P & \text{Restriction} & F ::= u \mid ?x \mid f(\tilde{F}) \quad \text{Pattern } (f\in\Sigma) \\
\mid\ !P & \text{Replication} &
\end{array}
$$

Fig. 1. Syntax of processes

$$
\begin{array}{rclrclrcl}
(P|Q)|R & \equiv & P|(Q|R) & (\nu n)(\nu m)P & \equiv & (\nu m)(\nu n)P & P|(\nu n)Q & \equiv & (\nu n)(P|Q) \text{ if } n\notin \mathrm{fn}(P) \\
P|Q & \equiv & Q|P & (\nu n)\mathbf{0} & \equiv & \mathbf{0} & ((\nu n)Q) > P & \equiv & (\nu n)(Q > P) \text{ if } n\notin \mathrm{fn}(P) \\
P|\mathbf{0} & \equiv & P & !P & \equiv & P|!P & r\rhd(\nu n)P & \equiv & (\nu n)(r\rhd P) \text{ if } r\neq n
\end{array}
$$

Fig. 2. Structural congruence

Structural congruence. Structural congruence $\equiv$ is defined as the least congruence relation induced by the laws in Figure 2. This set of laws comprises the structural rules for parallel composition and restriction from the π-calculus, plus the obvious extension of restriction's scope extrusion law to pipelines and sessions.

2.2 Operational Semantics

We let $\xrightarrow{\lambda}$ be the labelled transition relation induced by the rules in Figure 4. Labels λ have the syntax and informal meaning defined in Figure 3, where τ is the silent action. We call a *reduction* any silent transitions $P \xrightarrow{\tau} Q$. Note that we shall often abbreviate $(\nu n_1)\cdots(\nu n_l)\lambda$ as $(\nu\tilde{n})\lambda$ where $\tilde{n} = n_1,...,n_l$. We define $\mathrm{n}(\lambda)$, $\mathrm{fn}(\lambda)$ and $\mathrm{bn}(\lambda)$ as expected; in particular $\mathrm{bn}(s(r)) = \mathrm{bn}(\overline{s}(r)) = \{r\}$.

Service Definition and Invocation. Rule (DEF) describes the behaviour of a service definition: it says that a service $s.P$ is ready to establish a new session named r. Rule (CALL) describes the complementary behaviour of a service invocation $\overline{s}.Q$. Rule (SYNC) describes session creation as the result of $s.P$ and $\overline{s}.Q$ synchronizing and, in doing so, agreeing on a fresh session name r. The new session has two ends, one client's side where protocol Q is running and one at service's side where protocol P is running. A value produced by a concretion at one side can be consumed by an abstraction at the other side.

$$
\begin{array}{lll}
\lambda ::= \tau & \lambda_i ::= s(r) & \text{(service definition)} \\
\mid \lambda_i & \mid (V) & \text{(value consumption)} \\
\mid \lambda_o & \mid r:(V) & \text{(consumption within } r\text{)}
\end{array}
\qquad
\begin{array}{ll}
\lambda_o ::= \overline{s}(r) & \text{(service invocation)} \\
\mid (\nu\tilde{n})\langle V\rangle & \text{(value production)} \\
\mid (\nu\tilde{n})r:\langle V\rangle & \text{(production within } r\text{)} \\
\mid (\nu\tilde{n})\uparrow V & \text{(value return)}
\end{array}
$$

Fig. 3. Transition labels

$$(\textsc{Def})\ \frac{r \notin \mathrm{fn}(P)}{s.P \xrightarrow{s(r)} r \rhd P} \qquad (\textsc{Call})\ \frac{r \notin \mathrm{fn}(P)}{\overline{s}.P \xrightarrow{\overline{s}(r)} r \rhd P} \qquad (\textsc{Sync})\ \frac{P \xrightarrow{s(r)} P' \quad Q \xrightarrow{\overline{s}(r)} Q'}{P|Q \xrightarrow{\tau} (\nu r)(P'|Q')}$$

$$(\textsc{Out})\ \langle V\rangle P \xrightarrow{\langle V\rangle} P \qquad (\textsc{Ret})\ \langle V\rangle^{\uparrow} P \xrightarrow{\uparrow V} P \qquad (\textsc{In})\ \frac{\mathrm{match}(F,V) = \sigma}{(F)P \xrightarrow{(V)} P\sigma}$$

$$(\textsc{Sum})\ \frac{\pi_i P_i \xrightarrow{\lambda} P'}{\sum_{i\in I} \pi_i P_i \xrightarrow{\lambda} P'} \qquad (\textsc{S-In})\ \frac{P \xrightarrow{(V)} P'}{r \rhd P \xrightarrow{r:(V)} r \rhd P'} \qquad (\textsc{S-Out})\ \frac{P \xrightarrow{\langle V\rangle} P'}{r \rhd P \xrightarrow{r:\langle V\rangle} r \rhd P'}$$

$$(\textsc{S-Sync})\ \frac{P \xrightarrow{r:\langle V\rangle} P' \quad Q \xrightarrow{r:(V)} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad (\textsc{S-Ret})\ \frac{P \xrightarrow{\uparrow V} P'}{r \rhd P \xrightarrow{\langle V\rangle} r \rhd P'}$$

$$(\textsc{S-Pass})\ \frac{P \xrightarrow{\lambda} P'}{r \rhd P \xrightarrow{\lambda} r \rhd P'}\ r' \neq r \qquad \lambda ::= \overline{s}(r') \mid s(r') \mid \tau \mid r' : (V) \mid r' : \langle V\rangle$$

$$(\textsc{P-Sync})\ \frac{P \xrightarrow{\langle V\rangle} P' \quad Q \xrightarrow{(V)} Q'}{P > Q \xrightarrow{\tau} (P' > Q) \mid Q'} \qquad (\textsc{P-Pass})\ \frac{P \xrightarrow{\lambda} P'}{P > Q \xrightarrow{\lambda} P' > Q}\ \lambda \neq \langle V\rangle$$

$$(\textsc{Open})\ \frac{P \xrightarrow{\lambda} P' \quad n \notin \mathrm{bn}(\lambda)}{(\nu n)P \xrightarrow{(\nu n)\lambda} P'}\ n \in \mathrm{n}(V) \qquad \lambda ::= (\nu\tilde{n}')\langle V\rangle \mid (\nu\tilde{n}') \uparrow V \mid (\nu\tilde{n}')r : \langle V\rangle \qquad (\textsc{R-Pass})\ \frac{P \xrightarrow{\lambda} P' \quad n \notin \mathrm{n}(\lambda)}{(\nu n)P \xrightarrow{\lambda} (\nu n)P'}$$

$$(\textsc{Par})\ \frac{P \xrightarrow{\lambda} P'}{P|Q \xrightarrow{\lambda} P'|Q}\ \mathrm{fn}(Q) \cap \mathrm{bn}(\lambda) = \emptyset \qquad (\textsc{Struct})\ \frac{P \equiv Q \quad Q \xrightarrow{\lambda} Q' \quad Q' \equiv P'}{P \xrightarrow{\lambda} P'}$$

Fig. 4. Labelled Operational Semantics

Communication prefixes. Rule (OUT) models the behaviour of concretion $\langle V\rangle P$ that can evolve to P with a label $\langle V\rangle$, denoting emission of value V. Rule (IN) models the behaviour of an abstraction $(F)P$ that can be seen as a form of guarded command that relies on pattern-matching: $(F)P$ can evolve to $P\sigma$ with (V), indicating consumption of a value, only provided the pattern F matches up the value V. This leads to a substitution σ such that $\mathrm{match}(F,V) = \sigma$. Here, the pattern-matching function match is defined as expected: $\mathrm{match}(F,V) = \sigma$, if σ is the (only) substitution such that $\mathrm{dom}(\sigma) = \mathrm{bn}(F)$ and $F\sigma = V$. Rule (RET) models the behaviour of the return primitive $\langle V\rangle^{\uparrow} P$ that can be used to return a value *outside* the current session if the enclosing environment is capable of consuming it. Guarded choice has the expected meaning: $\sum_{i\in I} \pi_i P_i$ evolves with λ to P' if there is an $i \in I$ such that $\pi_i P_i$ evolves with λ to P' (see rule (SUM)).

Communication inside sessions. Rules (S-IN) and (S-OUT) add the name of the innermost enclosing session to the labels for intra-session communication. Rule (S-SYNC) finalizes communication inside session r for complementary action labels.

Communication outside sessions. Rule (S-RET) transforms a return performed inside a session r into the output of a value for the enclosing environment. Rule (S-PASS) propagates all session-transparent activities, namely $s(r')$, $\overline{s}(r')$, $r' : (V)$, $(\nu\tilde{n})r' : \langle V\rangle$ and τ. In

fact, services can be accessed and invoked independently from the hierarchy of sessions, and intra-session communication is always bound to the innermost enclosing session.

Pipelining. Rule (P-SYNC) expresses that in a pipeline $P > Q$ all the values V produced by P and that can be consumed by Q will trigger a new instance Q' of Q. (Note that, after this reduction, Q is again ready to consume the next value produced by P, if any.) Rule (P-PASS) indicates that the pipeline is transparent to all the other transitions of P, which are thus propagated to the enclosing context. Also note that Q is idle until a value produced by P activates one instance of it.

Restriction, parallel and structural congruence. Rules (OPEN), (R-PASS) and (PAR) are the standard ones for restriction and parallel. However, thanks to the rule for structural congruence (STRUCT), we need not a *close* rule for names extruded via rule (OPEN), because all steps can be performed in normalized processes, with all restrictions moved to the top.

Additional Comments. Sessions, service definitions and service invocations can of course be nested at arbitrary depth. Note that no activity can take place under the scope of a dynamic operator (see Definition 1 below). On the contrary, when considering static contexts (see Definition 2 below), concurrent activities can take place at any level of the session hierarchy. Also note that sessions are completely transparent with respect to actions different from value production/consumption/return, that is, service invocation and silent steps.

Remark 1 (About reduction semantics.). It has become common to present the operational semantics of newly proposed calculi by means of reductions instead of labelled transitions, with the advantage of a simpler and compact presentation. However, in the case of CaSPiS, the nesting and interplay of sessions and pipelines introduces some contextual dependencies on the reductions rules that makes the labelled transitions more appealing. The interested reader may check Lemma 3 to see all the different kinds of reductions that must be taken into account (the lemma is given for the full CaSPiS).

2.3 Well-Formedness

As anticipated, we introduce a well-formedness criterion to impose some discipline on the way CaSPiS primitives can be used and rule out many pathologically wrong process designs. To this aim, we first introduce some terminology and technical constraints. We say P *has no top bound sessions* if $P \equiv (\nu r)P'$ implies $r \notin \mathrm{fn}(P')$, for each P', r.

A *context* is a process term with one occurrence of a distinct process variable, say X (representing the "hole"). In what follows, we shall indicate by $C[\cdot]$ a generic context, and by $C[P]$ the process obtained when textually replacing X by P in $C[\cdot]$. The term Q *occurs in* (or is a subterm of) P if there is a context $C[\cdot]$ such that $P = C[Q]$. The notion of context can be generalized to n-holes contexts as expected. In particular, generic 2-holes contexts will be denoted by $C[\cdot,\cdot]$, with $C[P,Q]$ defined as obvious.

Definition 1. Dynamic *operators are service definition* $s.[\cdot]$ *and invocation* $\overline{s}.[\cdot]$*, any prefix* $\pi_i[\cdot]$*, right-hand side term of a pipeline* $P > [\cdot]$ *and replication* $![\cdot]$*. The remaining operators are* static.

Definition 2. *A context $C[\cdot]$ is* static *if its hole does not occur in the scope of a dynamic operator. Moreover, we say that $C[\cdot]$ is* session-immune *if its hole does not occur in the scope of a session operator.*

We introduce the final ingredient needed for our definition of well-formed process.

Definition 3. *Given P and two session names $r, r' \in \mathrm{fn}(P)$, we write $r \prec_P r'$ if and only if $P \equiv C[r \rhd C'[r' \rhd P']]$ for some static $C[\cdot]$, session-immune $C'[\cdot]$ and P'.*

In other words, $r \prec_P r'$ if, in P, up to structural congruence, an occurrence of r' is immediately within the scope of some session side $r \rhd [\cdot]$. Well-formedness is formally defined below:

Definition 4 (well-formedness). *Assume $P \equiv (\nu \tilde{r})Q$, where Q has no top bound sessions and $\tilde{r} \subseteq \mathrm{fn}(Q)$. Then, process P is* well-formed *if: (a) relation $\prec_Q$ is acyclic (that is, $\prec_Q^+$ is irreflexive), (b) modulo alpha conversion, for each r, $r\rhd$ occurs at most twice in P and never in the scope of a dynamic operator, (c) in any summation $\sum_i \pi_i$, all prefixes π_i are of one and the same kind (either all abstractions, or all concretions or all returns).*

The acyclicity of $\prec_Q$ rules out vicious situations like $r \rhd (P_1 | r \rhd P_2)$. For the rest, distinct service invocations, even of the same service, should give rise to distinct and fresh session names, and, of course, each session should normally have no more than two sides (one-sided sessions make sense once we allow one side to autonomously close, a scenario that we shall consider in Section 4). Also, for technical reasons, it is desirable to forbid mixed sums, that is sums where prefixes of different kinds occur.

In the remainder of this paper, *all processes are assumed to be well-formed*. By inspection, it is straightforward to check the validity of the following result.

Lemma 1. *Well-formedness is preserved by structural congruence and reductions.*

3 CaSPiS at Work

In this section we present some simple examples that aim at showing how CaSPiS can be used for specifying behaviours of structured services. The examples expose some important patterns for service composition, for which we find it convenient to introduce a set of derived operators.

In the following we will assume the following services are available. Service `emailMe` when invoked with argument *msg* has the effect of sending a message *msg* to one's email address. Services `ANSA`, `BBC` and `CNN`, upon invocation, return a possibly infinite sequence of values representing pieces of news (disregarding the identity of these news, these services resemble $!\texttt{ANSA}.!(\nu n)\langle n\rangle$, etc.).

Invocation patterns. Recurrent service invocation patterns are reported in Figure 5:

- $\overline{s}\langle V\rangle$ invokes the service s and then sends the value V over the established session;
- $\overline{s}(V)$ invokes s, then sends value V over the established session, then waits for a value from the service (the result of the service invocation) and publishes it locally (outside the established session);

$$\overline{s}\langle V\rangle \triangleq \overline{s}.\langle V\rangle \quad \text{(One Way)} \qquad P \gg \overline{s} \triangleq P > (?x)\overline{s}(x) \quad \text{(Simple Pipe)}$$
$$\overline{s}(V) \triangleq \overline{s}.\langle V\rangle(?x)\langle x\rangle^{\uparrow} \quad \text{(Request Response)} \qquad \overline{s}(!) \triangleq \overline{s}.!(?x)\langle x\rangle^{\uparrow} \quad \text{(Get All Responses)}$$

Fig. 5. CaSPiS derivable constructs

- $P \gg \overline{s}$ is a pipeline based composition to invoke service s on all values produced by P;
- $\overline{s}(!)$ invokes s, then keeps retrieving and publishing all responses from s.

Using the request-response and one-way patterns we can rewrite the example in the Introduction as: $\overline{\texttt{sign}}(\texttt{plan}) > (?z)\overline{\texttt{store}}\langle z\rangle$. Simple pipe can be used to write the signing of a document by two different authorities as $\langle \texttt{plan}\rangle \gg \overline{\texttt{sign}_1} \gg \overline{\texttt{sign}_2}$. Usage examples of the get-all-responses pattern are reported below for invoking news services.

Selection. Command **select** permits collecting the first n values emitted by a set of processes running in parallel and satisfying a given sequence of patterns.

Formally, we define **select** $F_1,\ldots,F_n$ **from** P as follows:

$$\textbf{select } F_1,\ldots,F_n \textbf{ from } P \triangleq (\nu s)\left(s.(F_1)\ldots.(F_n)\langle \hat{F}_1,\ldots,\hat{F}_n\rangle^{\uparrow} \mid \overline{s}.P\right)$$

where for each pattern F_i, $\hat{F}_i$ denotes the value V_i obtained from F_i by replacing each $?x$ with x. A useful variation of the above command is **select** $F_1,\ldots,F_n$ **from** P **in** Q, defined as follows:

$$\textbf{select } F_1,\ldots,F_n \textbf{ from } P \textbf{ in } Q \triangleq \textbf{select } F_1,\ldots,F_n \textbf{ from } P > (F_1,\ldots,F_n)Q$$

As an example the process

$$\textbf{select } ?x,?y \textbf{ from } \left(\overline{\texttt{ANSA}}(!) \mid \overline{\texttt{BBC}}(!) \mid \overline{\texttt{CNN}}(!)\right) \textbf{ in } \overline{\texttt{emailMe}}\langle x,y\rangle$$

will send to the specified email address a pair of the first two pieces of news among those arriving from ANSA, BBC and CNN, no matter who has actually produced them.

Waiting. When waiting for values produced by a set of concurrent activities, it may be useful not only to constrain the patterns of the expected values, but also to fix the exact binding between patterns and activities, that is, to specify which activity is expected to produce what. For instance, in the previous example, one might want to receive a mail with three pieces of news, the first coming from ANSA, the second from BBC and the third from CNN. This also implies that a mail will be sent only when a piece of news has been received from each of these services. We let **wait** $F_1,\ldots,F_n$ **from** $P_1,\ldots,P_n$ be a process that emits tuple $\langle V_1,\ldots,V_n\rangle$, where V_i is the first value emitted by P_i and matching F_i, for $i = 1,...,n$. We have that:

$$\begin{aligned}\textbf{wait } F_1,\ldots,F_n \textbf{ from } P_1,\ldots,P_n \triangleq (\nu s)\big(\; & s.(t_1(F_1))\ldots.(t_n(F_n))\langle \hat{F}_1,\ldots,\hat{F}_n\rangle^{\uparrow} \\ & \mid \overline{s}.(\; \textbf{select } F_1 \textbf{ from } P_1 \textbf{ in } \langle t_1(\hat{F}_1)\rangle \mid \cdots \\ & \quad \mid \textbf{select } F_n \textbf{ from } P_n \textbf{ in } \langle t_n(\hat{F}_n)\rangle\;)\;\big)\end{aligned}$$

We can also consider the following variation:

$$\textbf{wait}\ F_1,\ldots,F_n\ \textbf{from}\ P_1,\ldots,P_n\ \textbf{in}\ Q \triangleq \textbf{wait}\ F_1,\ldots,F_n\ \textbf{from}\ P_1,\ldots,P_n > (F_1,\ldots,F_n)Q$$

Then, **wait** $?x,?y,?z$ **from** $\overline{\texttt{ANSA}}(!)\,,\overline{\texttt{BBC}}(!)\,,\overline{\texttt{CNN}}(!)$ **in** $\overline{\texttt{emailMe}}\langle x,y,z\rangle$ will send an email containing the first pieces of news distributed by each of ANSA, BBC and CNN.

Travel Agent Service. We put at work macros *select* and *wait* defined above for describing a classical example of service composition: a Travel Agent Service. A travel agent offers its customers the ability to book packages consisting of services offered by various providers. For instance: Flight and Hotel Booking. Three kinds of booking are available: Flight only, Hotel only, or both. A customer contacts the service and then provides it with appropriate information about the planned travel (origin and destination, departing and returning dates,...). When a request is received, the service contacts appropriate services (for instance Lufth and Alit for flights, and HInn and BWest for hotels) and then compares their offers to select the most convenient (this is actually done by invoking a sub-service compare), which is returned to the customer.

This service can be specified in CaSPiS as follows:

$$\begin{array}{rl} !\texttt{ta}. & (\mathit{fly}(?x)).\textbf{wait}\ ?y,?z\ \textbf{from}\ \overline{\texttt{Lufth}}\langle x\rangle,\overline{\texttt{Alit}}\langle x\rangle\ \textbf{in}\ \overline{\texttt{compare}}(y,z) \\ + & (\mathit{hotel}(?x)).\textbf{wait}\ ?y,?z\ \textbf{from}\ \overline{\texttt{BWest}}\langle x\rangle,\overline{\texttt{HInn}}\langle x\rangle\ \textbf{in}\ \overline{\texttt{compare}}(y,z) \\ + & (\mathit{fly}\&\mathit{hotel}(?x)).\textbf{wait}\ ?y,?z\ \textbf{from}\ \overline{\texttt{ta}}(\mathit{fly}(x)),\overline{\texttt{ta}}(\mathit{hotel}(x))\ \textbf{in}\ \langle y,z\rangle \end{array}$$

After invocation, one message among three possible kinds of requests is expected: $\mathit{fly}(V)$, $\mathit{hotel}(V)$ and $\mathit{fly}\&\mathit{hotel}(V)$, where value V contains trip information. For example, the first case, two values, each containing an offer for a flight, are obtained from Lufth and Alit. The pair of these values is passed to service compare that selects the most convenient, and then immediately returned on to the client. Note that, upon receiving a *fly*&*hotel* request, service TA recursively invokes itself for determining the best offers for flight and hotel. These values are then returned to the client.

π-*calculus channels.* By analogy with SCC [3], it can be easily inferred that the close-free fragment of CaSPiS is expressive enough to model (lazy) λ-calculus and that, in the absence of pattern matching, it can be encoded in π-calculus.

The close-free fragment of CaSPiS is also expressive enough to encode π-calculus. Indeed, input and output over a channel a can be encoded in CaSPiS as follows:

$$a(x).P \triangleq a.(?x)\langle x\rangle^{\uparrow} > (?x)P \qquad \overline{a}v.P \triangleq \overline{a}.\langle v\rangle\langle\bullet\rangle^{\uparrow} > (\bullet)P$$

A proxy service. We conclude this section by considering the description of a simple *proxy service* that, once received a service name s and a value x, invokes s with parameter x and sends back to the caller all the values emitted by s:

$$!\mathit{proxy}.(?s,?x)\overline{s}.\langle x\rangle!(?y)\langle y\rangle^{\uparrow}$$

4 The Full Calculus

The calculus we have presented in the preceding sections offers no primitives for handling session closing. These primitives might be useful to garbage-collect terminated sessions. Most important, one might want to explicitly program session termination, in order to implement cancellation workflow patterns [24], or to manage abnormal events, or timeouts.

Sessions are units of client-server cooperation and as such their termination must be programmed carefully. At least, one should avoid situations where one side of the session is removed abruptly and leaves the other side dangling forever. Also, subsessions should be informed and, e.g., close in turn. The full CaSPiS we are going to introduce comprises mechanisms for programming disciplined closing sessions. The mechanism of session termination we shall adopt has been informally described in the Introduction. Before introducing this extension formally, we discuss a couple of additional issues below.

An important aspect of our modelling is that, when shutting down a side, the emitted signal $\dagger(k)$ is (realistically) *asynchronous*. In other words, we have no guarantee as to when the termination handler at the opposite side will be reached by $\dagger(k)$. So, for example, by the time $\dagger(k)$ reaches its destination, the other side might in turn have entered a closing state $\blacktriangleright Q$ on its own, or be closed right away, as a result of the closing of a parent session. These aspects must be taken into account when defining what "well-programmed" closing means (see Section 5). In general, dangling $\dagger(k)$ cannot be avoided, but we will be able to avoid sessions dangling forever.

It is worth mentioning that there are at least two obvious alternatives to the mechanism we have chosen. One would be to use close as a primitive for terminating instantaneously *both* the client-side and service-side sessions. But, as discussed above, this strategy conflicts with the two parties being in charge for the closing of their own session sides. A second alternative would be to use close as a synchronization primitive, so that the client-side and service-side sessions are terminated when close is encountered on one side and $\overline{\mathsf{close}}$ on the other side. This strategy conflicts with parties being able to decide autonomously when to end their own sessions. The use of termination handlers looks a reasonable compromise: each party can exit a session autonomously but it is obliged to inform the other party.

4.1 Syntax and Operational Semantics of the Full Calculus

Syntax. In what follows, we assume a new countable set $\mathcal{K}$ of *signal names* $k, k', \ldots$, disjoint from session and service names. The syntax of full CaSPiS is reported in Figure 6. In addition to the extended primitives $s_k.P$, $\overline{s}_k.P$ and $r \rhd_k P$ and to the new primitives close, $\dagger(k)$ and $\blacktriangleright P$ discussed in the Introduction, note the presence of termination listeners $k \cdot P$ that are used to handle termination signals $\dagger(k)$.

In what follows, we say that a name *n occurs free in P underneath a context* $C[\cdot]$ if there is a term Q such that $n \in \mathrm{fn}(Q)$ and $P = C[Q]$. For instance, k occurs free in $!k \cdot \mathbf{0}$ underneath $![\cdot]$, while it does not occur free in $!(\nu k)(k \cdot \mathbf{0})$ underneath $![\cdot]$.

Well-formedness. Beside those reported in Section 2, we assume the following additional well-formedness conditions on the syntax presented in Figure 6:

$$
\begin{array}{rlll}
P,Q ::= & \sum_{i\in I}\pi_i P_i & \text{Guarded Sum} & \mid \dagger(k) \quad \text{Signal} \\
\mid & s_k.P & \text{Service Definition} & \mid r \rhd_k P \quad \text{Session} \\
\mid & \overline{s}_k.P & \text{Service Invocation} & \mid \blacktriangleright P \quad \text{Terminated Session} \\
\mid & P > Q & \text{Pipeline} & \mid P|Q \quad \text{Parallel Composition} \\
\mid & \mathsf{close} & \text{Close} & \mid (\nu n)P \quad \text{Restriction} \\
\mid & k \cdot P & \text{Listener} & \mid\ !P \quad \text{Replication}
\end{array}
$$

Fig. 6. Syntax of full CaSPiS

$$
\begin{array}{ccc}
r \rhd_{k'} (\dagger(k)|P) \equiv \dagger(k)|r \rhd_{k'} P & (\dagger(k)|P) > Q \equiv \dagger(k)|(P > Q) & \blacktriangleright \dagger(k) \equiv \dagger(k) \\
\blacktriangleright r \rhd_k P \equiv \blacktriangleright r \rhd_k \blacktriangleright P & \blacktriangleright (P > Q) \equiv (\blacktriangleright P) > Q & \blacktriangleright\blacktriangleright P \equiv \blacktriangleright P \\
\blacktriangleright P|Q \equiv \blacktriangleright P|\blacktriangleright Q & \blacktriangleright (\nu x)P \equiv (\nu x)\blacktriangleright P & \blacktriangleright \mathbf{0} \equiv \mathbf{0}
\end{array}
$$

Fig. 7. Structural congruence rules for $\dagger(k)$ and $\blacktriangleright$

- for any signal name k and term P, modulo alpha-equivalence there is at most one subterm of P of the form $r \rhd_k Q$; moreover, k does not occur in concretion prefixes or return prefixes, and does not occur *free* in P underneath any dynamic context;
- Terminated sessions operators $\blacktriangleright$ do not occur within the scope of a dynamic operator.

The above conditions are easily seen to be preserved by the structural rules and by the SOS rules presented below. Note that the second condition above implies that passing of signal names is forbidden. In what follows, we assume all terms are well-formed.

Structural congruence. Nesting of sessions may give rise to subtle race conditions on the order of closings. As an example, consider a situation with two sessions r_1 and r_2, both ready to close and with r_2 nested in r_1:

$$r_1 \rhd_{k_1} \left(\ \mathsf{close} \ \mid \ P \ \mid \ r_2 \rhd_{k_2} (\mathsf{close}|Q) \ \right).$$

Suppose the innermost session r_2 closes up first; then, before the signal $\dagger(k_2)$ reaches its listener, also session r_1 closes up. This leads to the situation $\dagger(k_1) \mid \blacktriangleright (P \mid \dagger(k_2) \mid \blacktriangleright Q)$, where one still wants to activate the listener on k_2, despite the fact that $\dagger(k_2)$ lies within a terminated session. This example shows that terminated session operator, $\blacktriangleright$, should stop any activity *but* invocation of listeners, that is signals $\dagger(k)$.

The structural rules listed in Figure 7 enrich the set of rules already introduced for the basic calculus. The law $\blacktriangleright \dagger(k) \equiv \dagger(k)$ has been motivated above. The remaining rules serve the purpose of letting signals $\dagger(k)$ freely move within a term to reach the corresponding listeners, and distributing the terminated session $\blacktriangleright$ over static operators.

Note that, by structural congruence, in any term it is possible to move all restrictions not in the scope of a dynamic operator to top level.

Operational semantics. The SOS rules for the new operators are reported in Figure 8, the rules for the remaining operators are the same as those introduced for the basic

$$(\textsc{Def})\ s_{k_1}.P \xrightarrow{s(r)_{k_1}^{k_2}} r \rhd_{k_2} P \quad (\textsc{Call})\ \overline{s}_{k_2}.P \xrightarrow{\overline{s}(r)_{k_2}^{k_1}} r \rhd_{k_1} P \quad (\textsc{Sync})\ \frac{P \xrightarrow{s(r)_{k_1}^{k_2}} P' \quad Q \xrightarrow{\overline{s}(r)_{k_2}^{k_1}} Q'}{P|Q \xrightarrow{\tau} (\nu r)(P'|Q')}$$

$$(\textsc{Lis})\ k \cdot P \xrightarrow{k} P \qquad (\textsc{Sig})\ \dagger(k) \xrightarrow{\overline{k}} \mathbf{0} \qquad (\text{T-}\textsc{Sync})\ \frac{P \xrightarrow{k} P' \quad Q \xrightarrow{\overline{k}} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$(\textsc{End})\ \mathsf{close} \xrightarrow{\mathsf{close}} \mathbf{0} \quad (\text{S-}\textsc{End})\ \frac{P \xrightarrow{\mathsf{close}} P'}{r \rhd_k P \xrightarrow{\tau} \blacktriangleright P'|\dagger(k)} \quad (\text{T-}\textsc{End})\ \blacktriangleright r \rhd_k P \xrightarrow{\tau} \blacktriangleright P|\dagger(k)$$

Fig. 8. Labelled Operational Semantics of full CaSPiS

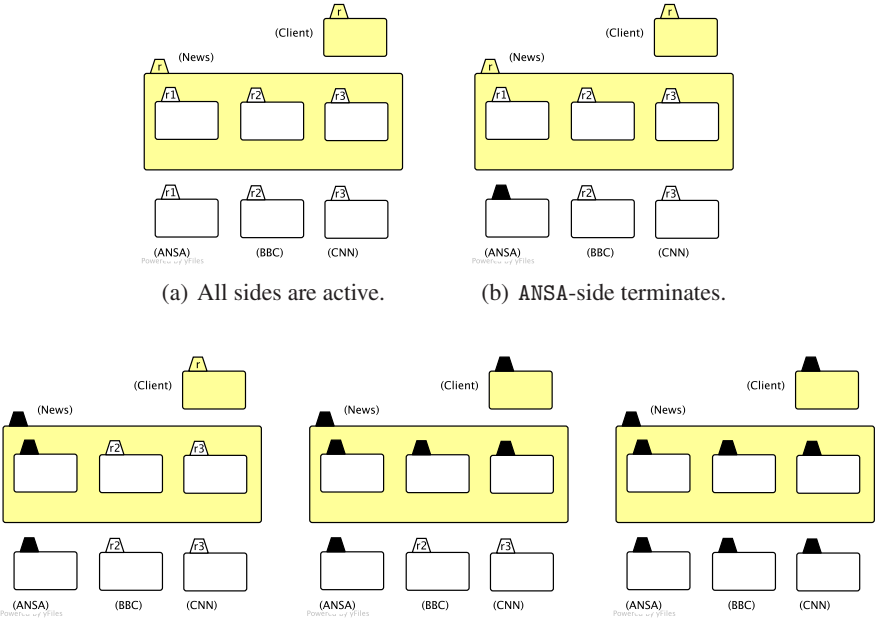
calculus (Section 2). Roughly: (1) rules (DEF), (CALL) and (SYNC) have been updated to take into account the exchange of termination handler names, (2) new rules (LIS), (SIG) and (T-SYNC) are used to synchronize a listener with a pending signal for it, (3) new rules (END) and (S-END) are used to close a session from inside and generate a signal towards the handler of the partner, and (4) a new rule (T-END) is used to recursively close session that were nested in a closed session (in a top-down fashion). Garbage collecting rules for guarded sums, service definitions and service invocations, like $\blacktriangleright \overline{s}_k.P \xrightarrow{\tau} \mathbf{0}$, are omitted for simplicity, but they can be added without any relevant consequence on the results in Section 5.

Example 1. Let us consider process *News* defined as follows:

$$\begin{aligned} \mathit{News} \stackrel{\triangle}{=} !(\nu k)\mathtt{collect}_k.\ \big(\ & k \cdot \mathsf{close} \mid (\nu k_1)\overline{\mathtt{ANSA}}_{k_1}.(!(?x)\langle x\rangle^{\uparrow} \mid k_1 \cdot (\mathsf{close}|\dagger(k))) \\ & \mid (\nu k_2)\overline{\mathtt{BBC}}_{k_2}.(!(?x)\langle x\rangle^{\uparrow} \mid k_2 \cdot (\mathsf{close}|\dagger(k))) \\ & \mid (\nu k_3)\overline{\mathtt{CNN}}_{k_3}.(!(?x)\langle x\rangle^{\uparrow} \mid k_3 \cdot (\mathsf{close}|\dagger(k)))\ \big) \end{aligned}$$

News specifies a *news collector* exposing service `collect`. After invocation of this service, a client receives all the news produced by `ANSA`, `BBC` and `CNN`. The established session can be closed: either (i) by the client-side, when an action close on the client's side is performed, as this will yield a signal $\dagger(k)$ able to activate the corresponding service-side listener $k \cdot \mathsf{close}$; or, (ii) when any of the three nested sessions used for interacting with the news services is closed by peer, yielding the signal $\dagger(k_i)$ and hence $\dagger(k)$. The termination of the topmost session will in turn cause the termination of all (not yet terminated) nested news clients.

Figure 9 shows a possible propagation of termination. Initially all services have been invoked and the corresponding sessions have been triggered (Figure 9(a)). Suppose the session running on `ANSA`-side shuts down (Figure 9(b)), then its partner session-side running as a child of the main session of the news collector is informed and terminated. Moreover, the listener causes the termination of the parent session (Figure 9(c)). Termination of the main session causes the termination of the remaining two children sections and of the main client-side session (Figure 9(d)). Finally, the sessions running on `BBC`-side and `CNN`-side are also terminated (Figure 9(e)).



(a) All sides are active. (b) `ANSA`-side terminates.

(c) `News`-side terminates. (d) Client-side terminates. (e) `BBC/CNN`-side terminate.

Fig. 9. Propagation of side-termination via listeners

5 Programming Graceful Termination

The session closing primitives introduced in the preceding section do not guarantee *per se* that forever-dangling, one-sided sessions never arise, in the same way as deadlock can arise in untyped π-calculus processes or termination cannot be guaranteed for all sequential programs. In this section, we show that this undesirable situation can be avoided if one makes sure that termination handlers of the form $k \cdot C[\mathsf{close}]$, for suitable static contexts $C[\cdot]$ and signals k, are installed in the bodies of client invocations and service definitions. We will be rather liberal with the choice of $C[\cdot]$, that may contain extra actions the termination handler may wish to take upon invocation, e.g., further signaling to other listeners (a sort of compensation, in the language of long-running transactions). For example, we anticipate that the process *News* from Example 1 fits our requirements.

The key to the proof is a concept of *balanced* term, roughly, a term with only pairs of session-sides that balance with each other. We shall prove that these terms enjoy what we call a "graceful" termination property: informally, *any possibly unbalanced term reachable from a balanced term can get balanced in a finite number of reductions*. Technically, this result will be achieved in two steps. First, we shall introduce a notion of *quasi*-balanced term, that generalizes that of balanced term. The key property here is that, differently from balanced-ness, quasi-balanced-ness is preserved through reductions. Next, we prove that any quasi-balanced term can reduce to a balanced one

in a finite number of reductions. In order to define these concepts, we need to introduce some terminology about contexts.

Updated vocabulary. We revisit the terminology presented in Section 2.3 to deal with the extended syntax of full CaSPiS. A context is *static* if the hole does not occur in the scope of a dynamic operator, and *quasi-static* if the hole does not occur in the scope of a dynamic operator, except possibly the dead-session operator $\blacktriangleright$. In what follows, $D[\cdot], D'[\cdot], \ldots$ will be used to denote generic quasi-static contexts. Note that reductions and execution of close actions are permitted underneath static context but, in general, not underneath quasi-static ones. We say that $C[\cdot]$ is *session-immune* if its hole is not in the scope of a session operator.

Main definitions. In order to ensure that graceful session closing at runtime, well-formedness does not suffice, and we have to restrict our syntax in certain ways. The first step is to ensure that session termination is properly programmed inside all service definitions and invocations. That is, both on client- and service-side, proper listeners are in place.

Definition 5 (graceful property). *We say P is* graceful *if, whenever $P \equiv Q$, Q satisfies the following conditions for each s and k: (a) $s_k.$ and $\overline{s}_k.$ may only occur in Q in subterms of the form $s_k.(C[k \cdot C'[\mathsf{close}]])$ or $\overline{s}_k.(C[k \cdot C'[\mathsf{close}]])$, with $C[\cdot], C'[\cdot]$ static and session immune; (b) in Q there is at most one occurrence of the listener $k\cdot$ and one occurrence of $\triangleright_k$.*

For example, obvious "graceful" usages for service invocation and service definition are $(\nu k_1)\overline{s}_{k_1}.(P_1 | k_1 \cdot \mathsf{close})$ and $(\nu k_2)s_{k_2}.(P_2 | k_2 \cdot \mathsf{close})$, respectively.

Lemma 2. *Let P be graceful and suppose $P \xrightarrow{\tau} Q$. Then Q is graceful.*

The proof of Lemma 2 involves some case analysis based on the following general lemma that allows us to identify active and passive components of any reduction $P \xrightarrow{\tau} Q$. (Note that, by definition, the graceful property is preserved by structural congruence.)

Lemma 3 (context lemma for reductions). *Suppose $P \xrightarrow{\tau} Q$. Then there is a 1- or 2-hole static context, $C[\cdot]$ or $C[\cdot,\cdot]$, such that one of the following cases is true (for some r, k, k', k'', s, V_i's, F_j's, P_i's, R_j's, R, P', σ, static and session-immune $C_0[\cdot], C_1[\cdot], C_2[\cdot]$ such that* $\mathrm{match}(V_l, F_h) = \sigma$*).*

1. *(Sync)* $P \equiv C[s_k.P, \overline{s}_{k'}.R]$, $Q \equiv (\nu r)C[r \triangleright_{k'} P,\ r \triangleright_k R]$ $\quad r$ *fresh for* $P, C[\cdot], R$
2. *(S-Sync)* $P \equiv C[r \triangleright_k (P' | \sum_i \langle V_i \rangle P_i),\ r \triangleright_{k'} C_0[\sum_j (F_j) R_j]]$, $Q \equiv C[r \triangleright_k (P' | P_l),\ r \triangleright_{k'} C_0[R_h\sigma]]$
3. *(S-Sync-Ret)* $P \equiv C[r' \triangleright_k \left(r \triangleright_{k''} C_1[\sum_i \langle V_i \rangle^{\uparrow} P_i] | P'\right),\ r \triangleright_{k'} C_2[\sum_j (F_j) R_j]]$, $Q \equiv C[r' \triangleright_k \left(r \triangleright_{k''} C_1[P_l] | P'\right),\ r \triangleright_{k'} C_2[R_h\sigma]]$

4. *(P-Sync)* $\begin{array}{l} P \equiv C[(\sum_i \langle V_i \rangle P_i | P') > C_1[\sum_j (F_j) R_j]] \\ Q \equiv C[R_h \sigma \,|\, \big((P_l | P') > C_1[\sum_j (F_j) R_j]\big)\,]\end{array}$

5. *(P-Sync-Ret)* $\begin{array}{l} P \equiv C[\big((r \rhd_k C_0[\sum_i \langle V_i \rangle^{\uparrow} P_i]) \,|\, P'\big) > C_1[\sum_j (F_j) R_j]] \\ Q \equiv C[R_h \sigma \,|\, \big((r \rhd_k C_0[P_l] | P') > C_1[\sum_j (F_j) R_j]\big)]\end{array}$

6. *(T-Sync)* $P \equiv C[\dagger(k) | k \cdot R]$ *and* $Q \equiv C[R]$
7. *(S-End)* $P \equiv C[r \rhd_k C_0[\mathsf{close}]]$ *and* $Q \equiv C[\blacktriangleright\, C_0[\mathbf{0}] | \dagger(k)]$
8. *(T-End)* $P \equiv C[\blacktriangleright\, (r \rhd_k R)]$ *and* $Q \equiv C[\blacktriangleright\, R | \dagger(k)]$

The graceful property is not sufficient to guarantee the main result about we are after, because it says nothing about balancing of existing sessions. In order to define balancing at the level of sessions, we need to introduce some more terminology.

Definition 6 (*r*-balancing). *Let P be a process. We say P is*

- r-balanced *if either $r \notin \mathrm{fn}(P)$ or, for some static $C[\cdot]$ not mentioning r, k and k' (with $k \neq k'$), $P \equiv C[r \rhd_k A, r \rhd_{k'} B]$ where one of the following holds for some static, session-immune $C'[\cdot]$ and $C''[\cdot]$*
  (a) $A \equiv C'[\mathsf{close}]$
  (b) $A \equiv C'[\dagger(k') | k' \cdot C''[\mathsf{close}]]$
  (c) $A \equiv C'[k' \cdot C''[\mathsf{close}]]$
  and either (a) *or* (b) *or* (c) *holds for B, with k in place of k'.*
- quasi r-balanced *if either P is r-balanced or, for some static $C[\cdot]$ not mentioning r, $P \equiv C[r \rhd_k A]$ with either of* (a) *or* (b) *above holding for A, or $P \equiv C[\blacktriangleright\, r \rhd_k A]$ with either of* (a) *or* (b) *or* (c) *above holding for A.*

We are now ready to give the definition of (quasi-)balancing for processes.

Definition 7 ((quasi-)balanced processes). *Assume $P \equiv (\nu \tilde{r}) Q$, where Q has no top bound sessions and $\tilde{r} \subseteq \mathrm{fn}(Q)$. We say P is* balanced *(resp.* quasi-balanced*) if:*

(a) P is graceful, and
(b) for each $r \in \mathrm{fn}(Q)$, Q is r-balanced (resp. quasi r-balanced).

Clearly balancing implies quasi-balancing.

Theorem 1 (graceful termination). *Let P be balanced. Whenever $P \xrightarrow{\tau}{}^{*} P'$ there exists a balanced process Q such that $P' \xrightarrow{\tau}{}^{*} Q$.*

Proof (Sketch). The proof involves two main steps:

- First, we show that for any quasi-balanced process R, if $R \xrightarrow{\tau} R'$, then R' is also quasi-balanced.
- Second, we prove that for any quasi-balanced process R there is a balanced process R'' such that $R \xrightarrow{\tau}{}^{*} R''$.

Since P is balanced by hypothesis, then it is also quasi-balanced. Therefore P' is quasi-balanced (by straightforward induction, using the first fact above) and we can apply the second fact above to deduce the existence of a suitable Q reachable from P'.

Example 2. It is easy to observe that process *News* defined in Example 1 is balanced. This guarantees that whenever a client closes the session established for interacting with service `collect`, eventually all the nested sessions will be closed. Moreover, if the sessions that interact with the news servers are closed, the main session will be closed and the client will be notified. In fact, the example shows a "graceful" usage pattern for closing the parent session after the unexpected termination of a child session.

6 Conclusion, Related Work and Future Work

We have presented CaSPiS, a core calculus for service-oriented applications. One of the key notions of CaSPiS is that of a session, which allows for the definition of arbitrarily structured interaction modalities between clients and services. The presentation of CaSPiS have included the full formalization of the operational semantics in the SOS style and a simple discipline to program graceful session termination. A number of examples witness the expressiveness of our proposal. A recent prototype implementation is also available [2].

The design of CaSPiS has been influenced by the π-calculus [21] and by Orc [22]. Indeed, one could say that CaSPiS combines the dataflow flavour of Orc (pipelines) with the name-scoping mechanisms of the π-calculus (sessions). There are important differences with these two languages, though, that in our opinion make CaSPiS worth studying on its own. There is no notion of a session in Orc: client-service interaction is strictly request-response. Complex interaction patterns can possibly be programmed in Orc by correlating sequences of independent service invocations via some ad-hoc mechanism (e.g. state variables). Asymmetric parallel in Orc offers a way for implementing, e.g., simple cancellation patterns. However, in Orc each site invocation can return at most one value, so that cancellation has a purely local effect. Our graceful termination improves on that mechanism by dealing with nested sessions and informing any side about the termination of its opposite side. Sessions and pipelines can possibly be encoded into π-calculus, but this would certainly cost the use of several levels of "continuation channels" to specify where the results produced by a session should be sent (i.e., whether to a father session, to a pipeline or to the surrounding environment). This would make the resulting code pretty awkward (see also [3]). As our examples show, sessions and pipelines are handy constructs that is worth having as first-class objects. This fact becomes even more evident when dealing with session termination, which has no (obvious) counterpart in the π-calculus.

CaSPiS evolved from SCC [3], because the original proposal turned out to be unsatisfactory in some important respects. In particular, SCC had no dedicated mechanism for orchestrating values arising from different activities, and only had a rudimentary mechanism for handling session termination, that would immediately kill the process (and its subprocess) executing the closing action, without activating any compensation action. The first problem motivated the proposal of a few evolutions of SCC. The one proposed in [16] is *stream*-oriented, in that values produced by sessions are stored into dedicated queues, accessible by their names. The one proposed in [10] has instead dedicated message passing primitives to model communication in all directions (within a session, from inside to outside and vice-versa). As seen, CaSPiS relies solely on the

concept of pipeline, but introduces pattern matching. More recently, a location-aware extension of SCC has also been proposed [6] that allows for the dynamic joining of multiparty sessions, but session termination policies are not addressed there.

A number of other proposals have been put forward, in the last couple of years, aiming, like us, at providing process calculi to support specification and analysis of services, see e.g. [17,9,19,11,25]. We do not enter into a detailed description of these proposals, which are based on rather different concepts and primitives, like *correlation sets*.

Developing safe client-service interactions requires some notion of compliance between conversation protocols. In this respect, the presence of pipelines and nested sessions makes the dynamics of a CaSPiS session quite complex and substantially different from simple type-regulated interactions as found in the π-like languages of, e.g. [14,15], or in the finite-state contract languages of [4,12,13]. Two recent contributions exploring the problem of compliance in the setting of CaSPiS are [1,8], whose type systems make evident the benefits of the concept of session. A type inference algorithm is proposed in [20].

Regarding future work, the impact of adding a mechanism of *delegation* deserves further investigation. In fact, delegation could be simply achieved by enabling session-name passing that is forbidden in the present version. However, the consequences of this choice on the semantics are at the moment not clear at all. We also plan to investigate the use of the session-closing mechanism for programming long-running transactions and related compensation policies in the context of web applications, in the vein e.g. of [7,18], and its relationship with the cCSP and the sagas-calculi discussed in [5].

Acknowledgments. We thank the members of the SENSORIA project involved in Workpackage 2, Core Calculi for Service Oriented Computing, for stimulating discussions on Services and Calculi. Lucia Acciai and Leonardo Gaetano Mezzina read the manuscript providing us with suggestions for improvements.

References

1. Acciai, L., Boreale, M.: A type system for client progress in a service-oriented calculus. In: Festschrift in Honour of Ugo Montanari, on the Occasion of His 65th Birthday. Lect. Notes in Comput. Sci., vol. 5065. Springer, Heidelberg (to appear, 2008)
2. Bettini, L., De Nicola, R., Loreti, M.: Implementing session-centered calculi with IMC. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 17–32. Springer, Heidelberg (2008)
3. Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V.T., Zavattaro, G.: SCC: a service centered calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 38–57. Springer, Heidelberg (2006)
4. Bravetti, M., Zavattaro, G.: A theory for strong service compliance. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 96–112. Springer, Heidelberg (2007)
5. Bruni, R., Butler, M., Ferreira, C., Hoare, T., Melgratti, H., Montanari, U.: Comparing two approaches to compensable flow composition. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 383–397. Springer, Heidelberg (2005)

6. Bruni, R., Lanese, I., Melgratti, H., Tuosto, E.: Multiparty sessions in SOC. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 67–82. Springer, Heidelberg (2008)
7. Bruni, R., Melgratti, H., Montanari, U.: Nested commits for mobile calculi: extending join. In: IFIP TCS 2004, pp. 367–379. Kluwer Academics, Dordrecht (2004)
8. Bruni, R., Mezzina, L.G.: Types and deadlock freedom in a calculus of services, sessions and pipelines (submitted, 2008)
9. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: SOCK: a calculus for service oriented computing. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 327–338. Springer, Heidelberg (2006)
10. Caires, L., Viera, H.T., Seco, J.C.: The conversation calculus: a model of service oriented computation. Technical Report TR DIFCTUNL 6/07, Univ. Lisbon (2007)
11. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)
12. Carpineti, S., Castagna, G., Laneve, C., Padovani, L.: A formal account of contracts for web services. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 148–162. Springer, Heidelberg (2006)
13. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. In: Proc. of POPL 2008, pp. 261–272. ACM Press, New York (2008)
14. Gay, S.J., Hole, M.J.: Types and subtypes for client-server interactions. In: Swierstra, S.D. (ed.) ESOP 1999. LNCS, vol. 1576, pp. 74–90. Springer, Heidelberg (1999)
15. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
16. Lanese, I., Martins, F., Ravara, A., Vasconcelos, V.T.: Disciplining orchestration and conversation in service-oriented computing. In: SEFM 2007, pp. 305–314. IEEE Computer Society Press, Los Alamitos (2007)
17. Laneve, C., Padovani, L.: Smooth orchestrators. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 32–46. Springer, Heidelberg (2006)
18. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
19. Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)
20. Mezzina, L.G.: How to infer finite session types in a calculus of services and sessions. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 216–231. Springer, Heidelberg (2008)
21. Milner, R., Parrow, J., Walker, J.: A Calculus of Mobile Processes, I and II. Information and Computation 100(1), 1–40, 41–77 (1992)
22. Misra, J., Cook, W.R.: Computation orchestration: A basis for wide-area computing. Journal of Software and Systems Modeling 6(1), 83–110 (2007)
23. Sensoria Project. Public web site, http://sensoria.fast.de/
24. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distributed and Parallel Databases 14(1), 5–51 (2003)
25. World Wide Web Consortium. Web Services Choreography Description Language Version 1.0. http://www.w3.org/TR/2005/CR-ws-cdl-10/

Modular Preservation of Safety Properties by Cookie-Based DoS-Protection Wrappers

Rohit Chadha, Carl A. Gunter, Jose Meseguer,
Ravinder Shankesi, and Mahesh Viswanathan

Dept. of Computer Science, University of Illinois at Urbana-Champaign

Abstract. Current research on verifying security properties of communication protocols has focused on proving integrity and confidentiality using models that include a strong Man-in-the-Middle (MitM) threat. By contrast, protection measures against Denial-of-Service (DoS) must assume a weaker model in which an adversary has only limited ability to interfere with network communications. In this paper we demonstrate a modular reasoning framework in which a protocol $\mathcal{P}$ that satisfies certain security properties can be assured to retain these properties after it is "wrapped" in a protocol $\mathcal{W}[\mathcal{P}]$ that adds DoS protection. This modular wrapping is based on the "onion skin" model of actor reflection. In particular, we show how a common DoS protection mechanism based on cookies can be applied to a protocol while provably preserving safety properties (including confidentiality and integrity) that it was shown to have in a MitM threat model.

1 Introduction

System security has many aspects, including secrecy, authentication, access control, availability, and many target sub-systems such as hardware, network protocols, operating system, application software, and so on. Security properties are typically verified of sub-systems, for example, by showing that a network protocol correctly uses cryptography to preserve a secret. But what we care about is the *end-to-end* security of the whole system, noting that it is a complex combination of its sub-systems. Modularity constructs and modular reasoning about properties are crucial to obtaining such end-to-end security guarantees. In this paper we present some modularity techniques and preservation results of this nature about two quite different kinds of sub-systems and properties, namely, an underlying communication protocol that may enjoy some *safety properties*, either related to security or to broader requirements, and a sub-system protecting against Denial of Service (DoS), which ensures some *availability properties*.

In actual practice, DoS protection mechanisms are not described and implemented in a modular way: typically an underlying protocol is changed in an ad-hoc way. For example, the TCP protocol can be defended against SYN attacks using SYN cookies [7], but requirements such as the need to assure compatibility with clients running diverse versions of TCP prevent a modular addition of this strategy. In particular, the TCP solution cannot be applied directly to other protocols, because it is inter-woven in a non-modular way with the specifics of TCP. Our first contribution in this paper is to consider a common DoS protection mechanism based on cookies, provided and

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 39–58, 2008.

described in a modular way as a generic "wrapper" that can be applied to an underlying protocol under minimal assumptions. In this way, a DoS protection mechanism becomes highly reusable and modular: one can develop its generic DoS wrapper once and for all. The wrapper then provides the desired DoS protection regardless of the underlying protocol it is applied to (under minimal assumptions): no changes to the underlying protocol are required. Specifically, we leverage ideas from distributed object reflection, where a distributed object can be wrapped by a "meta-object" that mediates its communication, with no required changes to the code of the underlying object. In our treatment we use a simplified version of the "onion skin" model of actor reflection [2], formalized as rewrite theories [12, 31]. We specify in rewriting logic and study in detail a generic wrapper for cookies like the strategy used in TCP SYN cookies. To illustrate this modularity, we sketch how it can be applied to a protocol like Internet Key Exchange (IKE) as was done in IKEv2. Although, our modular approach is applied to cookie mechanism in this paper, we believe that our techniques will apply to other DoS protection mechanisms such as those described in [19, 24].

Our second contribution is to prove that the system composition obtained by adding a cookie-based protection wrapper to a protocol preserves all the safety properties enjoyed by the original protocol. That is, the new availability properties enjoyed by the wrapped protocol are obtained *without losing* any of the safety properties. We obtain this result by specifying the given protocol, the wrapper, and the wrapped protocol as rewrite theories, and proving that: (1) a suitable stuttering simulation exists between the wrapped protocol and the original one; and (2) such simulations preserve all safety properties satisfied by the original protocol. The stuttering simulation constructed as a homomorphic map between the rewrite theories of the protocol with the cookie-wrapper and the original protocol essentially "forgets" the cookies used for DoS protection. We point out here that there is no simulation of the original protocol by the wrapped protocol. This is because the presence of the DoS protection necessarily implies that "malicious" service requests from illegitimate clients, which would be serviced in the absence of the DoS protection, must be ignored by the server . The dropping of the malicious service requests by the server wrapper also implies that we have to assume that the protocols are executed in a lossy environment, which allows the underlying protocol to simulate the dropping of the service requests by loss of messages.

Note that the availability properties enjoyed by the DoS protection mechanism (in this case the cookie mechanism) are, by definition, those of the wrapped protocol. That is, they are *emergent* properties of the corresponding wrapped composition, which typically did not exist in the original protocol. In this paper, we *assume that these availability properties are analyzed separately* by existing methods [1, 3, 27, 29]. The main result of our paper then ensures that: (i) if the underlying protocol, say $\mathcal{P}$, satisfies a set Γ of safety properties; and (ii) if the application $\mathcal{W}[\mathcal{P}]$ of the cookie wrapper $\mathcal{W}$ to protocol $\mathcal{P}$ satisfies a set Δ of availability properties, *then* $\mathcal{W}[\mathcal{P}]$ satisfies *both* Δ and the safety properties Γ.

In the case of security-related safety properties Γ, such as secrecy and authentication properties, enjoyed by a protocol $\mathcal{P}$, such properties are not just enjoyed by $\mathcal{P}$ itself: they are enjoyed by $\mathcal{P}$ *in the context* of a malicious environment that includes a Man-in-the-Middle (MitM) threat, which we can specify with a separate rewrite theory $\mathcal{I}$. That

is, the security-related safety properties Γ are satisfied by the *union* of theories $\mathcal{P} \cup \mathcal{I}$. To cover also these security-related safety properties, we prove a second version of our main theorem corresponding to safety properties that involve a MitM threat, against which the original protocol had been proved secure. We show that such an attacker cannot violate any of the already-proved safety properties for the cookie-wrapped extension of the protocol. Since the assumptions on the underlying protocol are really minimal, our result applies to the preservation of security-related safety properties for a wide range of cryptographic protocols. Therefore, proofs of such properties do not have to be redone after such protocols are subsequently hardened against DoS attacks by a cookie mechanism. Furthermore, since the MitM attacker in the paper is parametrized by an equational theory, our preservation result also applies to safety properties proved in the presence of a MitM attacker that can exploit algebraic properties of the cryptographic constructs used in protocol messages. We discuss one example application of our main result to a concrete protocol, namely, IKEv2 and its cookie mechanism.

The main technical challenge in proving the existence of the stuttering simulation in presence of the MitM threat is that the simulation can no longer be a homomorphic map that just "forgets" cookies. This is because the MitM attacker can intercept the cookies and generate new messages such as embedding the intercepted cookies within encrypted messages. Since the wrappers only perform a check of the accompanying cookies, these encrypted messages may lead to protocol executions which will not be captured by a simulation that forgets the cookies. This issue is resolved by exploiting the capability of the MitM attacker to generate cookies – cookie generation by the wrappers is simulated by the MitM attacker generating and storing new cookies.

The paper is organized into seven sections. Section 2 describes some preliminary concepts. Section 3 describes the underlying protocol semantics. Section 4 describes the stuttering simulation and preservation results for the cookie wrapper. Section 5 describes the MitM intruder model and how safety properties are preserved for the application of the wrapper. Section 6 discusses future work and Section 7 gives conclusions and sketches related work.

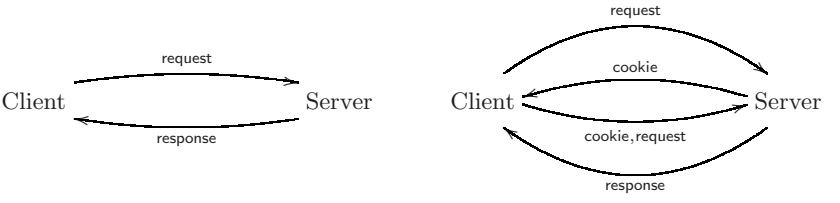
2 Preliminaries

The aim of this paper is to formalize, in a modular way, a simple cookie wrapper and to show that its use in transforming an underlying protocol does not invalidate the safety properties the underlying protocol enjoyed in a MitM model. The semantics of the underlying protocol and the protocol with the cookie wrapper are described using Kripke structures. A next-free safety fragment [35] of the logic LTL is used to specify safety properties. The preservation of safety properties is shown by exhibiting a stuttering simulation between the wrapped protocol and the underlying protocol. The Kripke structures are generated using rewrite theories.

2.1 Cookie-Based DoS Protection

A cookie is used in many network protocols (such as HTTP [15]), by one communicating party A to store some persistent information at B. A cookie k, when presented by

B to A, gives a weak guarantee that B has communicated with A at least once before to get the cookie. This guarantee is weak, because the cookie could be eavesdropped by a MitM intruder, who could impersonate B. However, in a typical (Distributed) Denial of Service flooding attack, the attacker tries to overwhelm a server's limited resources (such as memory or processing power) by sending, with little overhead, a large number of requests using different faked source IP address (see for e.g., [34]). If the cookie mechanism is used the attacker will not receive the cookies sent to the fake IP addresses unless it has a MitM ability. An adversary with MitM ability has already won, so DoS defense mechanisms target to protect against weaker adversaries.



For instance, a simple request-response protocol can be made DoS-resistant by using cookies and the communication pattern shown in the right of the above figure. The basic protocol is modified so that, when first contacted, the server sends a cookie to be stored at the client's side. The server simultaneously retains the ability to retrieve the cookie-value corresponding to that client. From then on, every client request is accompanied by the cookie, that the server can validate before committing any resources to that request. The server can reject all requests without the right cookies, with little overhead (see for e.g., [7]). Since the attacker uses spoofed addresses, it will not receive the cookie response from the server and thus cannot make any resource-intensive requests, thereby preventing the attack.

2.2 Kripke Structures, Safety Properties and Stuttering Simulation

The semantics of the underlying and wrapped protocols are defined using Kripke structures. Given a set of propositions AP, an AP-Kripke structure $\mathcal{A}$ is a tuple $(\mathsf{A}, \rightarrow_{\mathcal{A}}, \mathsf{L}_{\mathcal{A}})$ where the set A is the set of configurations of the protocol, the transition relation $(\rightarrow_{\mathcal{A}} \subseteq \mathsf{A} \times \mathsf{A})$ describes the temporal evolution of the configurations, and the labeling function $\mathsf{L}_{\mathcal{A}} : \mathsf{A} \rightarrow 2^{\mathsf{AP}}$ describes the set of propositions true in a configuration.

The safety-properties that we shall consider in this paper are expressed in the following next-free safety fragment [35] of the logic $\mathsf{LTL}(\mathsf{AP})$ (henceforth called $\mathsf{Safety}\backslash\bigcirc$). The syntax of the fragment $\mathsf{Safety}\backslash\bigcirc$ in BNF notation is:

$$\psi = (p) \;[]\; (\neg p) \;[]\; (\psi \vee \psi) \;[]\; (\psi \wedge \psi) \;[]\; (\psi \,\mathcal{W}\, \psi) \;[]\; (\Box\, \psi)$$

where $p \in \mathsf{AP}$. Here, the modalities $\mathcal{W}$ and $\Box$ are the usual weak-until and always modalities of LTL. We refer the reader to [35] for a formal definition. Please note that we could have also used other characterizations of safety properties such as the syntactic characterization using past operators [26] or the semantic characterization of safety [4].

As discussed above, we shall show that the safety properties are preserved by a wrapped protocol by exhibiting a stuttering simulation between the wrapped protocol

and the underlying protocol. Intuitively, a system $\mathcal{A}$ is simulated by a system $\mathcal{B}$ if every execution step of $\mathcal{A}$ can be matched by an execution step of $\mathcal{B}$. Since we are primarily interested in a next-free fragment of LTL, we require that an execution step of $\mathcal{A}$ is matched by zero or more execution steps of $\mathcal{B}$. Formally,

Definition: Given two AP-Kripke structures $\mathcal{A} = (\mathsf{A}, \rightarrow_{\mathsf{A}}, \mathsf{L}_{\mathsf{A}})$ and $\mathcal{B} = (\mathsf{B}, \rightarrow_{\mathsf{B}}, \mathsf{L}_{\mathsf{B}})$ a relation $\mathcal{H} \subseteq \mathsf{A} \times \mathsf{B}$ is said to be a *stuttering simulation* if for all a, b such that $a\mathcal{H}b$, the following hold: $\mathsf{L}_{\mathsf{A}}(a) = \mathsf{L}_{\mathsf{B}}(b)$, and if $a \rightarrow_{\mathcal{A}} a'$ then there exists a natural number $0 \leq j$ and elements $b_0, b_1, \ldots, b_j$ such that

1. $b_0 = b$
2. $b_l \rightarrow_{\mathcal{B}} b_{l+1}$ for all $0 \leq l < j$, and
3. there is a $0 \leq k \leq j$ such that $a\mathcal{H}b_l$ for all $0 \leq l \leq k$ and $a'\mathcal{H}b_l$ for all $k < l \leq j$.

The Kripke structure $\mathcal{A}$ is said to be stuttering simulated by $\mathcal{B}$ if there exists a stuttering simulation $\mathcal{H} \subseteq \mathsf{A} \times \mathsf{B}$.

Stuttering simulations reflect satisfaction of $\mathsf{Safety}\backslash\bigcirc$ properties.

Proposition 1. *Given two* AP*-Kripke structures* $\mathcal{A} = (\mathsf{A}, \rightarrow_{\mathsf{A}}, \mathsf{L}_{\mathsf{A}})$ *and* $\mathcal{B} = (\mathsf{B}, \rightarrow_{\mathsf{B}}, \mathsf{L}_{\mathsf{B}})$*, a stuttering simulation* $\mathcal{H} \subseteq \mathsf{A} \times \mathsf{B}$*, configurations* a, b *such that* $a\mathcal{H}b$ *and a next-free safety formula,* $\psi \in \mathsf{Safety}\backslash\bigcirc$*, we have* $\mathcal{B}, b \models_{\mathcal{B}} \psi \Rightarrow \mathcal{A}, a \models_{\mathcal{A}} \psi$*.*

2.3 Rewriting Logic

We specify the configurations of a protocol as the algebraic data type associated to an *order-sorted equational theory* (Σ, E) [16], where the signature Σ specifies the sorts, a subsort relation interpreted as subset inclusion in the algebras, and the constants and function symbols, and where E is a set of Σ-equations. The algebraic data type associated to (Σ, E) is the *initial algebra* $T_{\Sigma/E}$ [16]. In our protocols, the signature Σ will contain a sort Conf of object and message configurations as the chosen sort of configurations, so that the set of protocol configurations is the set $T_{\Sigma/E,\mathsf{Conf}}$. The theory (Σ, E) only specifies the *statics* of a protocol. A protocol $\mathcal{P}$, including its *dynamics*, is specified as a *rewrite theory* $\mathcal{P} = (\Sigma, E, R)$ [30], where the order-sorted equational theory specifies the configurations as explained above, and where R is a set of *rewrite rules* of the form $t \longrightarrow t'$ specifying the protocol concurrent transitions, that is, the protocol's "dynamics."

A set AP of atomic propositions for the configurations of a protocol $\mathcal{P} = (\Sigma, E, R)$ can be easily specified as equationally-defined Boolean predicates in (Σ, E). The atomic propositions AP, plus the choice of the sort Conf as the sort of configurations define a *Kripke structure* $\mathcal{K}(\mathcal{P})$ (see [9]), whose configurations are those of $\mathcal{P}$, whose transitions are the one-state rewrites with R modulo the equations E, and whose labeling function maps a state to all the atomic propositions that are provable true in that state. In this paper we allow deadlock configurations, and therefore the transition relation of $\mathcal{K}(\mathcal{P})$ is not required to be total.

In our modular reasoning we will use equational and rewrite theory *inclusions* $(\Sigma, E) \subseteq (\Sigma, E)$, and $(\Sigma, E, R) \subseteq (\Sigma, E, R)$, with the obvious meaning: $\Sigma \subseteq \Sigma'$, $E \subseteq E'$, and $R \subseteq R'$; and also theory *unions* $(\Sigma, E) \cup (\Sigma, E)$, and $(\Sigma, E, R) \cup (\Sigma, E, R)$, also with the obvious meaning: $\Sigma \cup \Sigma'$, $E \cup E'$, and $R \cup R'$.

3 Underlying Protocol

We use rewriting logic to express both an underlying, generic protocol satisfying minimal requirements, and the enhancement of such a protocol with DoS protection. The rewrite theories of both the underlying protocol and the enhancement of it with DoS protection shall assume that there is an equational theory $\mathcal{M} = (\Sigma_{\mathcal{M}}, E_{\mathcal{M}})$ which specifies the messages exchanged between clients and servers, with $\Sigma_{\mathcal{M}}$ a signature declaring sorts, subsorts and function symbols and $E_{\mathcal{M}}$ a set of Σ-equations. Furthermore, we make the following assumptions:

- There is a sort $\mathsf{MsgCnts}$ that describes the contents of a message. We hereafter use m, m_1, m_2, etc., as variables of sort $\mathsf{MsgCnts}$.
- There is a sort Cookie for cookies. The sort Cookie is a subsort of sort $\mathsf{MsgCnts}$. We use k, k', etc., as variables of sort Cookie.
- There is a function symbol $(\_,\_) : \mathsf{MsgCnts} \times \mathsf{MsgCnts} \rightarrow \mathsf{MsgCnts}$. Intuitively, the term (m_1, m_2) is the pair that consists of two messages m_1 and m_2.
- There is a sort Seed, a subsort of $\mathsf{MsgCnts}$, which is used by a unary function $\mathsf{rand} : \mathsf{Seed} \rightarrow \mathsf{Cookie}$, to generate a new cookie. There is a constant 1 of sort Seed and a unary function $\mathsf{next} : \mathsf{Seed} \rightarrow \mathsf{Seed}$ for incrementing the seed. As we shall see, the server wrapper stores a seed for cookie generation. The wrapper increments the seed each time it generates a fresh cookie. We shall use l, l' as variables of sort Seed. It will also be convenient to have a binary function symbol $\mathsf{Rnd} : \mathsf{Seed} \rightarrow \mathsf{MsgCnts}$ such that $\mathsf{Rnd}(l) = (\mathsf{next}(l), \mathsf{rand}(l))$.
- There are sorts CId and SId which stand for the identifiers for the clients and the servers. There is a sort Id with CId and SId as subsorts. The sort Id is a subsort of $\mathsf{MsgCnts}$. We use C, C_1, etc., as variables of sort CId, S, S_1, etc., as variables of sort SId and id, id_1, etc., to refer to variables of sort Id.
- There is a sort Msg and a ternary function symbol $(\mathsf{to}\ \_, \_\ \mathsf{from}\ \_) : \mathsf{Id} \times \mathsf{MsgCnts} \times \mathsf{Id} \rightarrow \mathsf{Msg}$. The term $(\mathsf{to}\ id_1, m\ \mathsf{from}\ id_2)$ stands for a message m sent by id_2 for id_1. We shall use msg, msg' as variables of sort Msg.
- There is also a constant $\mathsf{connect}$ of sort $\mathsf{MsgCnts}$. The term $(\mathrm{to\ S}, \mathsf{connect\ from}\ \mathrm{C})$ is the connect request sent by C to S.
- There is sort Conf along with a constant null of sort Conf. The sort Msg is a subsort of Conf. There is also a binary symbol function $\_\,;\,\_ : \mathsf{Conf} \times \mathsf{Conf} \rightarrow \mathsf{Conf}$ which intuitively stands for multiset union. Properties of associativity, commutativity and the identity (null) of $\_\,;\,\_$ are enforced in the set of equations $E_{\mathcal{M}}$. Later, we shall extend Conf to include the state of clients, servers and the intruder. A term of sort Conf will stand for protocol configurations and contain the messages, state of clients, servers and the intruder.

There might be other sorts and equations depending on the underlying protocol. For example, if the protocol uses symmetric encryption then there will be sorts for keys and two binary function symbols, say enc and dec, for encryption and decryption respectively. Furthermore, $E_{\mathcal{M}}$ will contain the equation $\mathsf{dec}(key, \mathsf{enc}(key, m)) = m$.

The messages on the network are represented as multisets of messages. Formally, a *multiset of messages* is a ground term of sort Conf defined recursively as follows:

1. If $\mathrm{id}_1, \mathrm{id}_2$ are ground terms of sort Id and m is a ground term of sort $\mathsf{MsgCnts}$ then $(\mathsf{to}\ \mathrm{id}_1,\ \mathrm{m}\ \mathsf{from}\ \mathrm{id}_2)$ is a multiset of messages.
2. If B_1 and B_2 are multisets of messages then $\mathrm{B}_1; \mathrm{B}_2$ is a multiset of messages.

For example, the multiset $(\mathsf{to}\ \mathrm{S}, \mathrm{m}\ \mathsf{from}\ \mathrm{C}); (\mathsf{to}\ \mathrm{C}, \mathrm{m}'\ \mathsf{from}\ \mathrm{S})$ represents two messages – m sent by (or some entity claiming to be) the client C meant for the server S and m' sent by (or some entity claiming to be) the server S meant for the client C.

Given the message equational theory $\mathcal{M} = (\Sigma_{\mathcal{M}}, E_{\mathcal{M}})$, the underlying protocol is given using a rewrite theory $\mathcal{P} = (\Sigma_{\mathcal{P}}, E_{\mathcal{P}}, R_{\mathcal{P}})$ with $\Sigma_{\mathcal{M}} \subseteq \Sigma_{\mathcal{P}}$ and $E_{\mathcal{M}} \subseteq E_{\mathcal{P}}$. The signature $\Sigma_{\mathcal{P}}$ in addition to $\Sigma_{\mathcal{M}}$ must contain the following sorts.

- There are sorts $\mathsf{CIntState}$ and $\mathsf{SIntState}$ which describe the internal states of the clients and servers respectively.
- There is a sort CConf, subsort of Conf, and a binary function symbol $\langle \_, \_ \rangle_c : \mathsf{CId} \times \mathsf{CIntState} \to \mathsf{CConf}$. We use X as variable of sort CConf. For example, the term $\langle \mathrm{C}, \mathrm{CIntState} \rangle_c$ represents a client whose unique identifier is C and whose internal state is $\mathrm{CIntState}$.
- There is a sort SConf, subsort of Conf, and a binary function symbol $\langle \_, \_ \rangle_s : \mathsf{SId} \times \mathsf{SIntState} \to \mathsf{SConf}$. We use Y as variable of sort SConf.

We shall assume that $\mathcal{P}$ ensures that the operator $(\_, \_) : \mathsf{MsgCnts} \times \mathsf{MsgCnts} \to \mathsf{MsgCnts}$ is a free constructor and that the operator $\_;\_ : \mathsf{Conf} \times \mathsf{Conf} \to \mathsf{Conf}$ is a free constructor modulo associativity, commutativity and identity of null.

We distinguish ground terms that represent configurations possibly reachable in a protocol execution. The reachable configurations consist of a multiset of messages, client states and server states. Furthermore, each client and server is represented by a unique term. The set of *good configurations* is defined inductively as follows.

- null is a good configuration.
- If Conf is a good configuration and B is a multiset of messages then $\mathrm{Conf}; \mathrm{B}$ is a good configuration.
- If Conf is a good configuration, C and $\mathrm{CIntState}$ are ground terms of sort CId and $\mathsf{CIntState}$ respectively, then $\mathrm{Conf}; \langle \mathrm{C}, \mathrm{CIntState} \rangle_c$ is a good configuration provided Conf does not have any subterm for the form $\langle \mathrm{C}, \mathrm{CIntState}' \rangle_c$ for some $\mathrm{CIntState}'$.
- If Conf is a good configuration, S and $\mathrm{SIntState}$ are ground terms of sort SId and $\mathsf{SIntState}$ respectively then $\mathrm{Conf}; \langle \mathrm{S}, \mathrm{SIntState} \rangle_S$ is a good configuration provided Conf does not have any subterm of the form $\langle \mathrm{S}, \mathrm{SIntState}' \rangle_s$ for some $\mathrm{SIntState}'$.

The set of good configurations shall henceforth be called $\mathsf{GoodConf}$. The execution of the protocol theory is given by a transition relation $\to_{\mathcal{P}}$. Given two good configurations Conf_1 and Conf_2, we write $\mathrm{Conf}_1 \to_{\mathcal{P}} \mathrm{Conf}_2$ iff Conf_2 can be obtained from Conf_1 by a one-step rewrite with a rule in $R_{\mathcal{P}}$.

For example, the configuration $\langle \mathrm{C}, \mathrm{CIntState} \rangle; \langle \mathrm{S}, \mathrm{SIntState} \rangle; (\mathsf{to}\ \mathrm{S}, \mathrm{m}\ \mathsf{from}\ \mathrm{C})$ represents a configuration in which a client C has sent a message m intended for server S. If the server reads the message m and changes its internal state to $\mathrm{SIntState}'$ then we shall have $\langle \mathrm{C}, \mathrm{CIntState} \rangle; (\mathsf{to}\ \mathrm{S}, \mathrm{m}\ \mathsf{from}\ \mathrm{C}); \langle \mathrm{S}, \mathrm{SIntState} \rangle \to_{\mathcal{P}} \langle \mathrm{C}, \mathrm{CIntState} \rangle; \langle \mathrm{S}, \mathrm{SIntState}' \rangle$.

4 Cookie Wrapper and Its Preservation Properties

Given the underlying protocol theory $\mathcal{P} = (\Sigma_{\mathcal{P}}, E_{\mathcal{P}}, R_{\mathcal{P}})$, the cookie wrapper is defined as a theory transformation $\mathcal{P} \mapsto \mathcal{W}[\mathcal{P}]$. The theory $\mathcal{W}[\mathcal{P}] = (\Sigma_{\mathcal{W}[\mathcal{P}]}, E_{\mathcal{W}[\mathcal{P}]}, R_{\mathcal{W}[\mathcal{P}]})$ extends $\mathcal{P}$ and is constructed as follows. The signature $\Sigma_{\mathcal{W}[\mathcal{P}]}$ extends $\Sigma_{\mathcal{P}}$ with the following new sorts:

1. There is a sort $\mathsf{CStoredCookiePair}$ along with a binary function symbol $(\_, \_) : \mathsf{SId} \times \mathsf{Cookie} \rightarrow \mathsf{CStoredCookiePair}$. Intuitively, the pair (S, k) will be stored by the client wrapper and it uses cookie k when sending requests to server S.
2. There is a sort $\mathsf{CStoredCookies}$ for the set of cookie pairs stored at the client site. There is a constant $\emptyset$ which stands for the empty set, along with a function symbol $\{\_\} : \mathsf{CStoredCookiePair} \rightarrow \mathsf{CStoredCookies}$ that make a term of sort $\mathsf{CStoredCookiePair}$ into a set and a function symbol $\cup$ that stands for the union operator. For example, the set $\{(\mathrm{S}, k)\} \cup \{(\mathrm{S}', k')\}$ stored at client site C means that the client C shall use the cookies k and k' while sending messages to S and S' respectively. There is also a binary function symbol $\mathsf{SIn} : \mathsf{SId} \times \mathsf{CStoredCookies} \rightarrow \mathsf{Bool}$. The function $\mathsf{SIn}(\mathrm{S}, \mathrm{CP}_c)$ returns true, if $\exists k$ s.t. $(\mathrm{S}, k) \in \mathrm{CP}_c$ and false otherwise. For example, the function $\mathsf{SIn}(\mathrm{S}_1, \{(\mathrm{S}, k)\} \cup \{(\mathrm{S}', k')\})$ returns true if and only if S_1 is either S or S'.
3. There are sorts $\mathsf{SStoredCookiePair}$ and $\mathsf{SStoredCookies}$ along with the binary function $\mathsf{CIn} : \mathsf{CId} \times \mathsf{SStoredCookies} \rightarrow \mathsf{Bool}$ for managing the cookies at server side similar to the ones at the client side.
4. There is a sort $\mathsf{WrappedCConf}$ for the wrapped client configuration along with a 4-ary function symbol $[\_\,, \_\,, \_\,, \_]_c : \mathsf{CId} \times \mathsf{CStoredCookies} \times \mathsf{Msg} \times \mathsf{CConf} \rightarrow \mathsf{WrappedCConf}$. The term $[\mathrm{C}, \mathrm{CP}_c, (\mathrm{to}\ \mathrm{S}, \mathrm{m}\ \mathrm{from}\ \mathrm{C}), X]$ stands for the client wrapper for the client C, where CP_c is the set of cookie pairs stored at client C, m is a message (destined for server S) that is stored while a connection to S is being established, and X is the underlying protocol configuration for the client C, together with messages sent by C or addressed to C.
5. There is a sort $\mathsf{WrappedSConf}$ along with a 4-ary function symbol $[\_\,, \_\,, \_\,, \_]_s : \mathsf{SId} \times \mathsf{SStoredCookies} \times \mathsf{Seed} \times \mathsf{SConf} \rightarrow \mathsf{WrappedSConf}$ for wrapped server configurations.
6. The sorts $\mathsf{WrappedSConf}$ and $\mathsf{WrappedCConf}$ are subsorts of Conf.

The set $E_{\mathcal{W}[\mathcal{P}]}$ extends $E_{\mathcal{P}}$ with equations required in the definition of new sorts. The set $R_{\mathcal{W}[\mathcal{P}]}$ extends $R_{\mathcal{P}}$ by adding new rules for the wrapper which are given in Table 1 and discussed below.

The client wrapper rule $\mathsf{ConnectReq}$ allows the wrapper to initiate a connection request to the server S if the connection is not already established and the underlying client wants to send a message M to S. The message M is held until the connection is established. The rule $\mathsf{SetCookie}$, triggered upon receiving a cookie from S, allows the wrapper to complete the connection and release M. The cookie is stored by the wrapper and the rule $\mathsf{MsgToServer}$ ensures that all future service requests to S contain this cookie. The rule $\mathsf{AcceptReply}$ allows the wrapper to forward any future replies from S to the underlying client.

Table 1. Cookie Wrapper Rewrite Rules

Client Wrapper Rules.

ConnectReq: $[C, CP_c, \mathsf{null}, (\mathsf{to}\ S, m\ \mathsf{from}\ C); X]_c \rightarrow [C, CP_c, (\mathsf{to}\ S, m\ \mathsf{from}\ C), X]_c;$
$(\mathsf{to}\ S, \mathsf{connect\ from}\ C)$ if $\mathsf{SIn}(S, CP_c) = \mathsf{false}$

SetCookie: $(\mathsf{to}\ C, k\ \mathsf{from}\ S); [C,\ CP_c, (\mathsf{to}\ S, m\ \mathsf{from}\ C)\ , X]_c \rightarrow$
$(\mathsf{to}\ S, (k, m)\ \mathsf{from}\ C); [C, \{(S, k)\} \cup CP_c, \mathsf{null}, X]_c$
if $\mathsf{SIn}(S, CP_c) = \mathsf{false}$

MsgToServer: $[C, \{(S, k)\} \cup CP_c, \mathsf{null}, (\mathsf{to}\ S, m\ \mathsf{from}\ C); X]_c \rightarrow$
$[C, \{(S, k)\} \cup CP_c, \mathsf{null}, X]_c; (\mathsf{to}\ S, (k, m)\ \mathsf{from}\ C)$

AcceptReply: $(\mathsf{to}\ C, m_1\ \mathsf{from}\ S); [C, CP_c, msg, X]_c \rightarrow$
$[C, CP_c, msg, (\mathsf{to}\ C, m_1\ \mathsf{from}\ S); X]_c$ if not $m_1 : \mathsf{Cookie}$

Service Wrapper Rules.

CookieGeneration: $(\mathsf{to}\ S, \mathsf{connect\ from}\ C); [S, CP_s, l, Y]_s \rightarrow (\mathsf{to}\ C, \mathsf{rand}(l)\ \mathsf{from}\ S)$
$[S, \{(C, \mathsf{rand}(\ l))\} \cup CP_s, \mathsf{next}(l), Y]_s$; if $\mathsf{CIn}(C, CP_s) = \mathsf{false}$

ResendCookie: $(\mathsf{to}\ S, \mathsf{connect\ from}\ C); [S, \{(C, k)\} \cup\ CP_s, l, Y]_s \rightarrow$
$[S, \{(C, k)\} \cup\ CP_s, l, Y]_s; (\mathsf{to}\ C, k\ \mathsf{from}\ S)$

ForwardRequest: $(\mathsf{to}\ S, (k, m)\ \mathsf{from}\ C); [S, \{(C, k)\} \cup CP_s, l, Y]_s \rightarrow$
$[S, \{(C, k)\} \cup\ CP_s, l, (\mathsf{to}\ S, m\ \mathsf{from}\ C);\ Y]_s$

DropRequest1: $(\mathsf{to}\ S, (k, m)\ \mathsf{from}\ C); [S, CP_s, l, Y]_s \rightarrow$
$[S, CP_s, l, Y]_s$ if $\mathsf{CIn}(C, CP_s) = \mathsf{false}$

DropRequest2: $(\mathsf{to}\ S, (k, m)\ \mathsf{from}\ C); [S, \{(C, k')\} \cup\ CP_s, l, Y]_s \rightarrow$
$[S, \{(C, k')\} \cup\ CP_s, l, Y]_s$ if $(k \neq k')$

ForwardReply: $[S, CP_s, l, (\mathsf{to}\ C, m\ \mathsf{from}\ S);\ Y]_s \rightarrow [S, CP_s, l, Y]_s; (\mathsf{to}\ C, m\ \mathsf{from}\ S)$

The server wrapper rule CookieGeneration, triggered when a server S receives a connection request from a client C for the first time, allows S to reply to the request with a freshly generated cookie. The cookie is stored and resent using the rule ResendCookie in reply to any further connection requests by C. The stored cookie is also used by rule ForwardRequest, which forwards service requests to S only if accompanied by the right cookie. The wrapper drops any service requests, if either the cookie mismatches or if there is no connection established, by using the rules DropRequest1 and DropRequest2 respectively. The rule ForwardRequest allows the wrapper to forward any client request (with the proper cookie) to the underlying server configuration. Finally, the rule ForwardReply allows the wrapper to forward any reply by its underlying server configuration to the client.

As in the case of a protocol theory, we need to identify the set of *good wrapped configurations* which represent the set of reachable configurations of the protocol with the DoS protection. In order to define good wrapped configurations we shall first define *well-formed unwrapped and wrapped client configurations*. A well-formed unwrapped client configuration for a client C is a ground term of sort Conf that contains a (unique) client configuration for C as well as messages sent by or sent to C. A well-formed wrapped client configuration for a client C consists of a ground term of

sort WrappedCConf which "wraps" a well-formed unwrapped client configuration for client C. Formally,

Definition: Given a ground term C of sort CId, a *well-formed unwrapped client configuration for* C is defined recursively as follows;

1. If Cis is a ground term of sort CIntState, then $\langle \mathrm{C}, \mathrm{Cis} \rangle_c$ is a well-formed unwrapped client configuration.
2. If Conf is well-formed unwrapped client configuration for C and m, S are ground terms of sort MsgCnts and SId respectively, then $\mathrm{Conf}; (\mathsf{to}\ \mathrm{S}, \mathrm{m}\ \mathsf{from}\ \mathrm{C})$ and $\mathrm{Conf}; (\mathsf{to}\ \mathrm{C}, \mathrm{m}\ \mathsf{from}\ \mathrm{S})$ are well-formed unwrapped client configurations.

If C,CP_c, S,m are ground terms of the sort CId, CStoredCookies, SId and MsgCnts respectively, and Conf is a well-formed unwrapped client configuration for C then $[\mathrm{C}, \mathrm{CP}_c, (\mathsf{to}\ \mathrm{S}, \mathrm{m}\ \mathsf{from}\ \mathrm{C}), \mathrm{Conf}]_c$ is a *well-formed wrapped client configuration*.

The well-formed unwrapped and wrapped server configurations can be defined analogously to their client counterparts.

We can now identify the set of good wrapped configurations which represent possible reachable configurations of the protocol with the cookie wrapper. A good wrapped configuration consists of a multiset of messages and well-formed wrapped client and server configurations with the restriction that any given client or server appears at most once. For lack of space, we do not provide a formal definition here. The set of good wrapped configurations shall henceforth be called GoodWrappedConf.

As before, we define the execution of the wrapped protocol by a one-step transition relation $\rightarrow_{\mathcal{W}[\mathcal{P}]} \subseteq \mathsf{GoodWrappedConf} \times \mathsf{GoodWrappedConf}$. Given two good wrapped configurations W_1 and W_2, we write $W_1 \rightarrow_{\mathcal{W}[\mathcal{P}]} W_2$ if W_2 can be obtained from W_1 by a one-step rewrite with a rule in $\mathcal{W}[\mathcal{P}]$. For example, the good wrapped configuration $[\mathrm{C}, \emptyset, (\mathsf{to}\ \mathrm{S}, \mathrm{m}\ \mathsf{from}\ \mathrm{C}), X]_c$; $(\mathsf{to}\ \mathrm{S}, \mathsf{connect}\ \mathsf{from}\ \mathrm{C})$; $[\mathrm{S}, \emptyset, 1, Y]_s$ represents a configuration in which the client wrapper for C has sent a connect request to the server S. The server accepts the connect request using the rewrite rule CookieGeneration and hence we have $[\mathrm{C}, \emptyset, (\mathsf{to}\ \mathrm{S}, \mathrm{m}\ \mathsf{from}\ \mathrm{C}), X]_c$; $(\mathsf{to}\ \mathrm{S}, \mathsf{connect}\ \mathsf{from}\ \mathrm{C})$; $[\mathrm{S}, \emptyset, 1, Y]_s \rightarrow_{\mathcal{W}[\mathcal{P}]}$ $[\mathrm{C}, \emptyset, (\mathsf{to}\ \mathrm{S}, \mathrm{m}\ \mathsf{from}\ \mathrm{C}), X]_c$; $(\mathsf{to}\ \mathrm{C}, \mathsf{rand}(1)\ \mathsf{from}\ \mathrm{S})$; $[\mathrm{S}, \{(\mathrm{C}, \mathsf{rand}(1))\}\ , \mathsf{next}(1)\ , Y]_s$.

We are almost ready to show that each execution of the wrapped protocol can be stuttering simulated by the unwrapped one. We shall however require one more definition. Consider the server wrapper rules DropRequest1 and DropRequest2. The server wrapper drops service requests if the cookie attached to the request mismatches the cookie stored at its site. This is not reflected in the underlying protocol as there is no such check of the service requests. However, if we consider protocols in the presence of a lossy environment, we can mimic the dropping of these requests by a message loss.

Given the equational theory $\mathcal{M} = (\Sigma_{\mathcal{M}}, E_{\mathcal{M}})$, a *lossy environment* is defined as the rewrite-theory $\mathcal{L} = (\Sigma_{\mathcal{L}}, E_{\mathcal{L}}, R_{\mathcal{L}})$, where $\Sigma_{\mathcal{L}} = \Sigma_{\mathcal{M}}$, $E_{\mathcal{L}} = E_{\mathcal{M}}$ and $R_{\mathcal{L}}$ consists of the single rewrite rule: $(\mathsf{to}\ id_1, m\ \mathsf{from}\ id_2) \rightarrow \mathsf{null}$. Similar to the way we have defined $\rightarrow_{\mathcal{P}}$ over GoodConf and $\rightarrow_{\mathcal{W}[\mathcal{P}]}$ over GoodWrappedConf, we can define the transition relations $\rightarrow_{\mathcal{P} \cup \mathcal{L}}$ over GoodConf and $\rightarrow_{\mathcal{W}[\mathcal{P}] \cup \mathcal{L}}$ over GoodWrappedConf.

4.1 Simulation

We shall show that the safety properties of the underlying protocol are preserved when we add the cookie-based DoS protection wrapper by giving a stuttering simulation map H_{fgt}: GoodWrappedConf $\rightarrow$ GoodConf between the set of good wrapped configurations GoodWrappedConf and the set of good configurations GoodConf.

The key idea in the construction of the simulation map H_{fgt} is that the connect requests and messages with cookies are used only by the DoS wrappers, and are not observed by the underlying protocols. The simulation map uses the auxiliary function h (given below), which maps a multiset of messages onto another multiset of messages by essentially forgetting all messages whose contents are only cookies or connect requests. Furthermore, in case the message consists of a cookie as the first part of the contents, the map h also forgets the cookie part. Formally,

$$
\begin{array}{ll}
\text{(a) } h((\mathsf{to}\ id_1, k\ \mathsf{from}\ id_2)) & = \mathsf{null} \\
\text{(b) } h((\mathsf{to}\ id_1, (k, m)\ \mathsf{from}\ id_2)) & = (\mathsf{to}\ id_1, m\ \mathsf{from}\ id_2) \\
\text{(c) } h((\mathsf{to}\ id_1, \mathsf{connect}\ \mathsf{from}\ id_2)) & = \mathsf{null} \\
\text{(d) } h(M) & = M \text{ if } M \text{ is not (a), (b) or (c)} \\
\text{(e) } h(M; X) & = h(M)\ ;\ h(X)
\end{array}
$$

Please note that we should read the above equations as working on equivalence classes of messages. For example, the equation $h((\mathsf{to}\ id_1, k\ \mathsf{from}\ id_2)) = \mathsf{null}$ means that any message which can be proved equal to $(\mathsf{to}\ id_1, k\ \mathsf{from}\ id_2)$ (using the set of equations E_p) also gets mapped to null. The well-definedness of equations will follow from the fact that the function $(\_, \_)$: MsgCnts $\times$ MsgCnts $\rightarrow$ MsgCnts is a free constructor and the function $\_; \_$: Conf $\times$ Conf $\rightarrow$ Conf is a free constructor modulo associativity, commutativity and identity of null.

We are now ready to define our simulation map H_{fgt}. Please note that our description of GoodWrappedConf implies that the good wrapped configurations are multisets which contain well-formed wrapped client configurations, well-formed wrapped server configurations, and a multiset of messages. The function H_{fgt} "unwraps" the client and server configurations, and maps the multisets of messages B to $h(B)$. The messages that are held in the client wrapper waiting for the server reply are "released".

Definition: Given the set of good configurations GoodConf and good wrapped configurations GoodWrappedConf, the function H_{fgt}: GoodWrappedConf $\rightarrow$ GoodConf is defined as follows.

$$
\begin{array}{l}
H_{fgt}\ (\ [C^1, CP_c^1, M^1, X_c^1]_c; \ldots; [C^q, CP_c^q, M^q, X_c^q]_c; \\
\qquad [S^1, CP_s^1, l^1, X_s^1]_s; \ldots; [S^r, CP_s^r, l^r, X_s^r]_s; B) = \\
\qquad\qquad X_c^1, \ldots X_c^q; X_s^1, \ldots X_s^r; h(B); M^1; \ldots; \ldots M^q
\end{array}
$$

For example, the configuration $[\mathrm{C}, \emptyset, (\mathsf{to}\ \mathrm{S}, \mathrm{m}\ \mathsf{from}\ \mathrm{C}), X]_c$; $(\mathsf{to}\ \mathrm{S}, \mathsf{connect}\ \mathsf{from}\ \mathrm{C})$; $[\mathrm{S}, \emptyset, 1, Y]_s$ in which the client wrapper for C has sent a connect request to the server S gets mapped by H_{fgt} to $X; (\mathsf{to}\ \mathrm{S}, \mathrm{m}\ \mathsf{from}\ \mathrm{C}); Y$.

Please recall that the transition systems (GoodConf, $\rightarrow_{\mathcal{P}}$) and (GoodWrappedConf , $\rightarrow_{\mathcal{W}[\mathcal{P}]}$) describe the evolution of unwrapped and wrapped protocol configurations respectively. In order to talk about safety properties, we need to define Kripke structures.

Towards this end, we shall assume that there is a set of propositions AP and a labeling function $L_{\mathcal{P}} : \mathsf{GoodConf} \rightarrow 2^{\mathsf{AP}}$ which labels the good configurations of the underlying protocol.[1]

Theorem 1 (Stuttering simulation and safety preservation). *Let $\mathcal{P}$ be a protocol rewrite theory and let $\mathcal{W}[\mathcal{P}]$ be the rewrite theory obtained by adding the cookie wrapper. Let $\mathcal{L}$ be the rewrite theory of a lossy environment. Let $\mathsf{A}_{\mathcal{P}\cup\mathcal{L}} = (\mathsf{GoodConf}, \rightarrow_{\mathcal{P}\cup\mathcal{L}}, \mathsf{L}_{\mathcal{P}\cup\mathcal{L}})$ be the Kripke structure generated from the good configurations of $\mathcal{P} \cup \mathcal{L}$ and $\mathsf{A}_{\mathcal{W}[\mathcal{P}]\cup\mathcal{L}} = (\mathsf{GoodWrappedConf}, \rightarrow_{\mathcal{W}[\mathcal{P}]\cup\mathcal{L}}, \mathsf{L}_{\mathcal{W}[\mathcal{P}]\cup\mathcal{L}})$ be the Kripke structure generated from the good wrapped configurations of $\mathcal{W}[\mathcal{P}] \cup \mathcal{L}$ such that $\mathsf{L}_{\mathcal{W}[\mathcal{P}]\cup\mathcal{L}} = H_{fgt} \circ \mathsf{L}_{\mathcal{P}\cup\mathcal{L}}$.*

Then $\{(W, H_{fgt}\ (W)) | W \in \mathsf{GoodWrappedConf}\}$ is a stuttering simulation. Furthermore, given a $\mathsf{Safety}\backslash\bigcirc$ formula ψ and any $W \in \mathsf{A}_{\mathcal{W}[\mathcal{P}]\cup\mathcal{L}}$,

$$H_{fgt}\ (W) \models_{\mathsf{A}_{\mathcal{P}\cup\mathcal{L}}} \psi \ \Rightarrow\ W \models_{\mathsf{A}_{\mathcal{W}[\mathcal{P}]\cup\mathcal{L}}} \psi.$$

Proof. (Sketch.) We have to show that if $W_1 \rightarrow_{\mathcal{W}[\mathcal{P}]\cup\mathcal{L}} W_2$ then it is matched by a stuttering transition of $H_{fgt}\ (W_1)$.

The transition $W_1 \rightarrow_{\mathcal{W}[\mathcal{P}]\cup\mathcal{L}} W_2$ can be obtained from one of the three theories: (i) a one-step application of a rewrite rule in the set $R_{\mathcal{P}}$ of the underlying protocol theory, (ii) an application of the new transition rules in $R_{\mathcal{W}[P]}$, and (iii) an application of the message drop rule in $\mathcal{L}$. In the first and third cases, the transition is trivially matched. For the second case, the rules ConnectReq, MsgToServer, SetCookie, AcceptReply, ResendCookie, ForwardRequest, CookieGeneration and ForwardReply can be shown to stutter.

The more interesting cases are the rules DropRequest1 and DropRequest2. In these cases the server wrapper drops messages if the cookies mismatch (or the server doesn't have any cookie corresponding to the client). This is simulated by a message loss transition of the environment.

Formally, if W_2 is obtained from W_1 by the application of either DropRequest1 or DropRequest2, then the multiset W_1 contains a server configuration $[S, CP_s, l, SConf]_s$ and a message $t_1 = (\mathsf{to}\ S, (k, m)\ \mathsf{from}\ C)$. Furthermore, (C, k) is not in the set CP_s. The wrapped configuration W_2 is obtained from W_1 by replacing t_1 by null. By definition, $W_1 = W'; t_1$, $W_2 = W'$, $H_{fgt}\ (W_1) = H_{fgt}\ (W'); H_{fgt}\ (t_1)$ and $H_{fgt}\ (W_2) = H_{fgt}\ (W')$. Now $H_{fgt}\ (t_1)$ is a message. Due to the message loss rule in $\mathcal{L}$, we get $H_{fgt}\ (W_1) \rightarrow_{\mathcal{P}\cup\mathcal{L}} H_{fgt}\ (W_2)$. □

Please note that, due to the absence of an intruder, this result doesn't apply to cryptographic security properties of the protocol. However, it is still useful as we may be concerned with other correctness properties of the protocol expressed as safety properties. This theorem ensures that any such properties regarding the protocol are still valid when we apply the cookie wrapper.

[1] Such atomic propositions and labeling function can, for example, be equationally defined in a conservative ("protecting") extension of the protocol theory $\mathcal{P}$ (see for e.g., [9]).

5 Security with a Man-in-the-Middle Intruder

The standard assumptions of a MitM model give the intruder the ability to intercept messages, store messages, and send messages on the network. It also has the ability to apply cryptographic functions available to the protocol 'users' (for instance decrypting an encrypted message using a key it already knows). We now give the rewrite theory for a Dolev-Yao intruder, a common MitM model, and later prove a stuttering simulation between a protocol and its cookie-based wrapped version, in the presence of this intruder, ensuring the preservation of safety properties. Please note that the Dolev-Yao intruder considered herein is parametrized by an equational theory which implies that our preservation result also applies to safety properties in the presence of an intruder that can exploit algebraic properties of the constructs (such as xor and modular exponentiation) used in protocol messages.

5.1 Rewrite Theory for the Dolev-Yao Intruder

Given the equational theory $\mathcal{M} = (\Sigma_{\mathcal{M}}, E_{\mathcal{M}})$, the capabilities of the Dolev-Yao intruder are defined in terms of a rewrite theory $\mathcal{I} = (\Sigma_{\mathcal{I}}, E_{\mathcal{I}}, R_{\mathcal{I}})$. The signature $\Sigma_{\mathcal{I}}$ extends $\Sigma_{\mathcal{M}}$ and the equations $E_{\mathcal{I}}$ extend $E_{\mathcal{M}}$, with the following additional sorts and equations (recall that we already have defined sorts Conf, MsgCnts and Seed as well as symbols next, rand, Rnd in the message equational theory $\mathcal{M}$).

- There is sort SetMsgCnts along with a constant null of sort SetMsgCnts. The sort MsgCnts is a subsort of SetMsgCnts. The sort SetMsgCnts stands intuitively for a set of terms of the sort MsgCnts. There is a binary function symbol $\_;\_ : \mathsf{SetMsgCnts} \times \mathsf{SetMsgCnts} \rightarrow \mathsf{SetMsgCnts}$ which intuitively stands for set union.
- There is a sort IntruderKnowledge, subsort of Conf, along with a unary function symbol $[\_]_i : \mathsf{SetMsgCnts} \rightarrow \mathsf{IntruderKnowledge}$. Intuitively, the term $[X]_i$ stands for the intruder's knowledge, *i.e.*, the intruder knows the set of terms X. For example, the term $[key, \mathrm{m}, \mathsf{enc}(key, m)]_i$ represents an intruder who knows a plain-text message m, a key key and the encryption of m under the key key.
- There is a set of function symbols $\mathcal{F}$ which represent the cryptographic functionality available to the intruder. $\mathcal{F}$ is assumed to contain at least the 0-ary function symbol connect, and the unary function symbol $\mathsf{Rnd} : \mathsf{Seed} \rightarrow \mathsf{MsgCnts}$ which satisfies the equation $\mathsf{Rnd}(l) = (\mathsf{next}(l), \mathsf{rand}(l))$ as before.

The rewrite rules $R_{\mathcal{I}}$ given in Table 2 describe the actions that the Dolev-Yao intruder can perform. The set $E_{\mathcal{I}}$ contains the equations $E_{\mathcal{M}}$, equations that make SetMsgCnts a set, and the equation Coupling-Decoupling given in Table 2. Furthermore, $E_{\mathcal{I}}$ may contain any equations that express the algebraic properties of functions in $\mathcal{F}$.

The equation Coupling-Decoupling allows the intruder to decompose a pair into its parts and allows pairs to be formed using two terms of sort MsgCnts. The rule MsgCreation allows the intruder to send messages. The rule MsgInterception allows the intruder to intercept messages, and the rule MsgDrop allows the intruder to drop a message without storing any knowledge about its contents. Finally, the rule CryptoFunctionality allows the intruder to compute the cryptographic functions $f \in \mathcal{F}$.

Table 2. Dolev-Yao Equations and Rules

Equation.
Coupling-Decoupling $[(m_1, m_2);\, Y]_i = [m_1; m_2;\, Y]_i$

Rewrite rules.

MsgCreation :	$[m; id_1; id_2; Y]_i \rightarrow [m; id_1; id_2;\, Y]_i; (\mathsf{to}\ id_1, m\ \mathsf{from}\ id)$
MsgInterception:	$[Y]_i; (\mathsf{to}\ id_1, m\ \mathsf{from}\ id) \rightarrow [m;\, Y]_i$
MsgDrop:	$[Y]_i; (\mathsf{to}\ id_1, m\ \mathsf{from}\ id) \rightarrow [Y]_i$
CryptoFunctionality:	$[m_1; \ldots; m_n;\, Y]_i \rightarrow [f(m_1, \ldots, m_n); m_1; \ldots; m_n;\, Y]_i$ if $f \in \mathcal{F}$

Please note that the Dolev-Yao intruder is powerful enough to exploit any algebraic properties specified in the equations $E_{\mathcal{I}}$.

$\mathcal{P} \cup \mathcal{I}$ and $\mathcal{W}[\mathcal{P}] \cup \mathcal{I}$ represent the rewrite-theories for the protocol and cookie-wrapped version of the protocol extended by the Dolev-Yao intruder. Analogous to GoodConf and GoodWrappedConf defined in Section 3, we represent reachable configurations in these extended protocols by using sets GoodExtConf and GoodExtWrappedConf. Intuitively, an element of GoodExtConf stands for a possible reachable configuration of the unwrapped protocol; and is a ground term of sort Conf that represents, client and server configurations, the messages on the network, and intruder knowledge represented by a term $[I]_i$ of sort IntruderKnowledge. We assume that the identifies of all clients and servers are included in the knowledge of the intruder. Similarly, an element of GoodExtWrappedConf represents a reachable configuration of the wrapped protocol in the presence of the Dolev-Yao intruder, again represented by a term of sort IntruderKnowledge. Furthermore, we shall require that the cookie-stores in the client, server wrappers are "well-formed". That is, the cookie-stores of each client, server wrappers contain a multiset of cookie pairs of the type (Id, k), where k is a ground term of sort Cookie. We shall also require that the seed in each server is of the form, $\mathrm{l} = \mathsf{next}^n(1)$, *i.e.*, n successive applications of the function next on constant 1.

5.2 Simulation

We shall now show that the safety properties of a protocol in the presence of the Dolev-Yao intruder are preserved when we add the cookie-based DoS protection by exhibiting a stuttering simulation $H_{\mathcal{I}Sim}$: GoodExtWrappedConf $\rightarrow$ GoodExtConf between the set of good extended configurations GoodExtConf and the set of good extended wrapped configurations GoodExtWrappedConf.

Please note that the simulation map H_{fgt} used in Theorem 1, which simply drops cookies from messages, will not suffice for our purpose. This is because we need to simulate all the actions that the intruder can do when cookie-based DoS protection is being used. In particular, the intruder in the wrapped theory (say I_w represented by a ground term of sort IntruderKnowledge) can intercept messages with cookies and then generate new messages with the intercepted cookies. For example, I_w can embed the

intercepted cookie within some encrypted message. The intruder in the unwrapped theory (say I_u, a ground term of sort IntruderKnowledge), however does not have access to these cookies, since there is no cookie generation by the server in the unwrapped protocol. However, the intruder I_u can use its cryptographic functionality Rnd to generate cookies, which allows us to simulate all of I_w's transitions.

The simulation map, $H_{\mathcal{I}Sim}$, is defined using four auxiliary functions. The first auxiliary function, h, is the same as the one defined earlier in Section 4. The second function, ck, maps a multiset of messages to a term of sort SetMsgCnts. Given a message M, the function ck picks out a cookie if either the message contents of M is just the cookie or if the message contents of M is a pair, the first component of which is a cookie. The other functions ck_c and ck_s similarly collect cookies in CStoredCookies and SStoredCookies by picking out the cookie part of every cookie pair stored in CStoredCookies, SStoredCookies. They are formally defined as follows:

$$
\begin{array}{ll}
\text{(a) } ck((\mathsf{to}\ id_1, k\ \mathsf{from}\ id_2)) & = k \\
\text{(b) } ck((\mathsf{to}\ id_1, (k, m)\ \mathsf{from}\ id_2)) & = k \\
\text{(c) } ck(M) & = \mathsf{null\ if\ M\ is\ not\ (a)\ or\ (b)} \\
\text{(d) } ck(M; X) & = ck(M)\,;\, ck(X) \\
\text{(e) } ck_c(\emptyset) & = null \\
\text{(f) } ck_c(\{(C, k)\} \cup CP_c) & = \{k\} \cup ck_c(CP_c) \\
\text{(g) } ck_s(\emptyset) & = null \\
\text{(h) } ck_s(\{(S, k)\} \cup CP_s) & = \{k\} \cup ck_s(CP_s)
\end{array}
$$

We are now ready to define our simulation map $H_{\mathcal{I}Sim}$. Please note that our description of GoodExtWrappedConf implies that the good extended wrapped configurations are multisets which contain well-formed wrapped client configurations, well-formed wrapped server configurations, an intruder state, and a multiset of messages. The function $H_{\mathcal{I}Sim}$ "unwraps" the client and server configurations, and maps the multiset of messages B to $h(B)$. The messages that are held in the client wrapper waiting for the server reply are "released". Finally, the intruder state in $H_{\mathcal{I}Sim}\ (W)$ has more facts than the intruder state in W, which allows it to mimic all actions of the intruder in W. In addition to all the facts available to the intruder in W, $H_{\mathcal{I}Sim}\ (W)$ also has cookies stored at client and server wrappers and in messages over the network. Please note that we are not giving the intruder access to the storage at these wrappers (there are no wrappers in $H_{\mathcal{I}Sim}\ (w)$). The condition is trivially satisfied at an "initial" state where the cookie stores are empty. The definition allows us to avoid an induction on reachable configurations. Furthermore, the intruder also has all the seeds that the server wrapper has, along with all the seeds that the server might have used in the past.

Definition: Given the set of good extended configurations GoodExtConf and the set of good extended wrapped configurations GoodExtWrappedConf; the function $H_{\mathcal{I}Sim}$: GoodExtWrappedConf $\rightarrow$ GoodExtConf is defined as follows.

$$
\begin{array}{l}
H_{\mathcal{I}Sim}\ (\,[C^1, CP_c^1, M^1, X_c^1]_c; \ldots; [C^q, CP_c^q, M^q, X_c^q]_c; \\
\qquad [S^1, CP_s^1, l^1, X_s^1]_s; \ldots; [S^r, CP_s^r, l^r, X_s^r]_s; [Y]_i; B) = \\
\quad X_c^1 \ldots X_c^q; X_s^1, \ldots X_s^r; h(B); M^1; \ldots; \ldots M^q \\
\qquad [Y; 1; \mathsf{next}(1); \mathsf{next}^2(1) \ldots; l^1; \ldots; 1; \mathsf{next}(1); \mathsf{next}^2(1) \ldots l^r; \\
\qquad\qquad ck_c(CP_c^1); \ldots; ck_c(CP_c^q); ck_s(CP_s^1); \ldots; ck_s(CP_s^r); ck(B)]_i.
\end{array}
$$

For example, consider the configuration $\mathrm{Conf} = [\mathrm{C}, \{\mathrm{S}, k\}, \mathsf{null}, X_c]_c; [\mathrm{S}, \{(\mathrm{C}, k)\}, \mathsf{next}(1), X_s]_s; (\mathsf{to}\ \mathrm{S}, (k, \mathrm{m})\ \mathsf{from}\ \mathrm{C})\ ; [\mathrm{C}, \mathrm{S}]_i$ in which the client has established a connection with the server S (with cookie k) and has sent a service request. The intruder has the identities of the client and the server. By definition, $H_{\mathcal{I}Sim}$ "unwraps" the client, the server, the service request, and stores the cookie k in the intruder memory. Formally, $H_{\mathcal{I}Sim}\ (\mathrm{Conf}) = X_c; X_s; (\mathsf{to}\ \mathrm{S}, \mathrm{m}\ \mathsf{from}\ \mathrm{C}); [\mathrm{C}, \mathrm{S}, 1, \mathsf{next}(1), k]_i$.

We have the second result of the paper.

Theorem 2. *Let $\mathcal{P} \cup \mathcal{I}$ be the rewrite theory for the protocol $\mathcal{P}$ in the presence of a Dolev-Yao intruder, and $\mathsf{A}_{\mathcal{P}\cup\mathcal{I}} = (\mathsf{GoodExtConf}, \rightarrow_{\mathcal{P}\cup\mathcal{I}}, \mathsf{L}_{\mathcal{P}\cup\mathcal{I}})$ be the Kripke structure generated from the good extended configurations. Let $\mathcal{W}[\mathcal{P}] \cup \mathcal{I}$ be the rewrite theory for the cookie wrapped protocol $\mathcal{W}[\mathcal{P}]$ in the presence of a Dolev-Yao intruder, and $\mathsf{A}_{\mathcal{W}[\mathcal{P}]\cup\mathcal{I}} = (\mathsf{GoodExtWrappedConf}, \rightarrow_{\mathcal{W}[\mathcal{P}]\cup\mathcal{I}}, \mathsf{L}_{\mathcal{W}[\mathcal{P}]\cup\mathcal{I}})$ be the Kripke struture generated from the good extended wrapped configurations, such that, $L_{\mathcal{W}[\mathcal{P}]\cup\mathcal{I}} = H_{\mathcal{I}Sim} \circ \mathsf{L}_{\mathcal{P}\cup\mathcal{I}}$.*

Then for any $\mathsf{Safety}\backslash\bigcirc$ *formula ψ and any $W \in \mathsf{A}_{\mathcal{W}[\mathcal{P}]\cup\mathcal{I}}$,*

$$H_{\mathcal{I}Sim}\ (W) \models_{\mathsf{A}_{\mathcal{P}\cup\mathcal{I}}} \psi \ \Rightarrow\ W \models_{\mathsf{A}_{\mathcal{W}[\mathcal{P}]\cup\mathcal{I}}} \psi.$$

Proof. (Sketch.) It suffices to show that $\{(W, H_{\mathcal{I}Sim}\ (W)) | W \in \mathsf{GoodWrappedConf}\}$ is a stuttering simulation, *i.e.*, whenever, $W_1 \rightarrow_{\mathcal{W}[\mathcal{P}]\cup\mathcal{I}} W_2$, then it is matched by a stuttering transition of $H_{\mathcal{I}Sim}\ (W_1)$.

The transition $W_1 \rightarrow_{\mathcal{W}[\mathcal{P}]\cup\mathcal{I}} W_2$ can either be a transition obtained by a one-step application of a rewrite rule of the underlying protocol theory $\mathcal{P}$, or a transition obtained by an application of the new transition rules of the wrapper $\mathcal{W}$, or a transition obtained by an application of the transition rules in the intruder theory. In the first case, the transition is trivially matched. For the second case, the rules ConnectReq, MsgToServer, SetCookie, AcceptReply, ResendCookie, ForwardRequest and ForwardReply can be shown to stutter. The rules DropRequest1, DropRequest2 can be simulated by the MsgDrop rule of the intruder.

If W_2 is obtained from W_1 by the application of CookieGeneration resulting in some server wrapper generating a cookie k, then this step is matched by the intruder using the its crypto functionality Rnd to generate the same cookie k.

Now, assume that W_2 is obtained from W_1 by the application of a rule in $\mathcal{R}_{\mathcal{I}}$. Please observe that the set of facts that the intruder possesses in W_1 is contained in the set of facts that the intruder possesses in $H_{\mathcal{I}Sim}\ (W_1)$. Hence, "crypto functionality" can be simulated exactly. Message creation of M can be simulated by "message creation" of $h(M)$. Please note that $h(M)$ creation requires less information. The "message interception" of M is simulated by interception of $h(M)$. The missing information (cookie) is already in the possession of the intruder in $H_{\mathcal{I}Sim}\ (W_1)$. □

5.3 Modular Reasoning with Cookies in IKEv2

In this section we describe how the cookie-wrapper can be applied to a practical protocol vulnerable to a DoS attack. Using the results in this paper, we can show that any $\mathsf{Safety}\backslash\bigcirc$ properties that have been verified on the original protocol (in the presence of a Dolev-Yao intruder) still apply to the cookie-wrapper version of the protocol (in

the presence of a Dolev-Yao intruder). In effect, we get the added benefit of some DoS protection without having to prove earlier safety properties again.

IPSec is a suite of protocols used to provide authentication and encryption on top of IP. The Internet Key Exchange (IKE) protocol is a part of the IPSec suite that allows two communicating parties to, among other things, securely generate session keys as well as exchange cryptographic parameters. IKEv2 ([23]) is the current version of the protocol. Since IPSec runs on top of the *connectionless* protocol UDP, there is no guarantee that an initiation request for an IKE session from a client has come from a valid IP address. This creates a DoS vulnerability as an attacker can send a large number of IKE initiation requests to a server with spoofed IP addresses. In fact, an earlier version of IKE (IKEv1[21, 28, 33]) was susceptible to this attack and protecting against such an attack was part of the motivation for IKEv2.

AVISPA ([5]) is an automated tool for verification of Internet protocols and applications. It has been used to verify various security protocols in the IPSec suite including the variants of IKEv2. In particular, it has been used to prove the secrecy of session keys exchanged in the key-exchange sub-protocol (IKEv2-DS[5]) which is vulnerable to a DoS attack. It follows from our results in this paper that the cookie-enhanced version of this protocol has the same guarantees.[2]

6 Related Work

Modular specification and reasoning has attracted a lot of attention in formal analysis of distributed systems and communication protocols. The advantage of this approach is to allow verification of individual sub-systems without worrying about *all* interactions between different layers. Towards this end, the "onion skin" modular model of actor reflection has been proposed in [2, 12, 31]. Our formalization of the cookie-based DoS protection mechanism can be seen as an instance of this formalism in which we are trying to develop the program we discussed in [3]. Ultimately we would like to be able to provide a formal modular treatment of protocols like Adaptive Selective Verification (ASV) based on the shared channel model [19, 24].

The modular framework for reasoning about a stack of protocols for security properties is less common. The security protocols are generally specified and analyzed individually. Recently, a modular proof of correctness (under Dolev-Yao assumptions) of the IEEE 802.11i and TLS suite of protocols was given in [22] using the Protocol Composition Logic [13]. Formal analysis of the VoIP protocol stack is carried out in [20] and the tunnel calculus is used to model security tunnels in stacks in [18]. Finally, there is a large body of work on compositional reasoning of cryptographic protocols. The primary focus of this line of research is to show that if some protocol is secure under the Dolev-Yao assumption of black-box cryptography, then the protocol remains secure when the black-box cryptographic functions are instantiated with their cryptographic realizations (see, *e.g.* [6]). Compositional reasoning is also the technique used in Protocol Composition Logic [10, 13], where, the correctness of a single protocol is derived by

[2] Note that IKEv2 has its own cookie-based mechanism for prevention of such an attack which has a slightly different implementation. Therefore, our claim applies to a modular wrapper-based implementation of cookies.

proving assertions about individual actions of honest participants. These compositional reasoning principles focus on safety properties and therefore complement the modular reasoning methods explored in this paper, which focus on the preservation of safety properties when DoS protection mechanisms are added.

The availability properties of the cookie-based mechanism is not the focus of our work, but there are several works in the literature which analyze availability properties. A general cost-based framework for analyzing protocols for DoS resistance has been proposed in [29] and is based on the fail-stop model in [17]. DoS-resistance is stated using an information flow property in a cost-based framework in [25]. Observation equivalence and a cost-based framework is used in [1] to analyze the availability properties of the JFK protocol. Other works on formal analysis of availability properties use branching-time logics [27, 36] and continuous stochastic logic [3].

Finally, we observe that rewriting logic has a well-established tradition in specification and analysis of security protocols [8, 11, 14, 32].

7 Conclusions and Future Work

We have proposed a modular approach to both the specification of DoS protection mechanisms as generic wrappers, and the preservation of safety properties of protocols when hardened by such wrappers. We have discussed how a cookie-based mechanism can be specified this way; and how safety properties of the underlying protocol are preserved by the cookie wrapper with and without the presence of a Dolev-Yao intruder.

This work represents a step toward capabilities we described in [3]. In that work we showed how a DoS-hardened protocol could be formally analyzed for its DoS protection capabilities by examining probabilistic guarantees for selective verification [19], a DoS protection mechanism based on the use of bandwidth limitations. The current work shows how to prove invariant properties between the MitM model and a model for DoS protection. However, the target DoS protection model is non-probabilistic (based on cookies). A next step along this path is to show this invariance for a system in which the DoS protection mechanism is probabilistic as in [3]. As an intermediate step, the results presented here about preservation of safety properties should be extended to preservation of properties definable in the more general logic $ACTL^*$ minus the next operator. Another intermediate step is to more completely formalize a protocol like IKEv2, so that the results of the paper can be more completely applied. A more ambitious topic worth investigating is the verification of *generic availability properties* of a given wrapper. That is, given a wrapper $\mathcal{W}$, can we reason about the availability properties that $\mathcal{W}$ will provide for *any* protocol $\mathcal{P}$ under minimal assumption on $\mathcal{P}$? Yet another topic for future research is the development of automated formal reasoning methods for proving preservation of a given class of properties of an underlying protocol $\mathcal{P}$ when such a protocol is composed with a given wrapper $\mathcal{W}$.

Acknowledgments. The authors thank Catherine Meadows, Omid Fatemieh, and Fariba Khan for their suggestions. We also benefited from comments by anonymous reviewers. Rohit Chadha was supported in part by NSF CCF04-29639 and NSF CCF04-48178. Carl Gunter was supported in part by NSF CNS05-24516 CNS05-5170 CNS05-09268

CNS05-24695 CNS07-16421, DHS 2006-CS-001-000001, ONR N00014-04-1-0562 N00014-02-1-0715, and a grant from the MacArthur Foundation. Jose Meseguer was supported in part by NSF CNS05-24516 and NSF CNS07-16638. Ravinder Shankesi was supported in part by NSF CNS05-24516. Mahesh Viswanathan was supported in part by NSF CCF04-48178 and NSF CCF05-09321. Views expressed in the paper are those of the authors.

References

1. Abadi, M., Blanchet, B., Fournet, C.: Just Fast Keying in the Pi Calculus. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 340–354. Springer, Heidelberg (2004)
2. Agha, G., Frolund, S., Panwar, R., Sturman, D.: A Linguistic Framework for Dynamic Composition of Dependability Protocols. IEEE Parallel and Distributed Technology: Systems and Applications 1, 3–14 (1993)
3. Agha, G., Greenwald, M., Gunter, C.A., Khanna, S., Meseguer, J., Sen, K., Thati, P.: Formal modeling and analysis of dos using probabilistic rewrite theories. In: Foundations of Computer Security (FCS 2005) (2005)
4. Alpern, B., Schneider, F.B.: Defining Liveness. Information Processing Letters 21(4), 181–185 (1985)
5. Armando, A., et al.: The Avispa Tool for the Automated Validation of Internet Security Protocols and Applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 281–285. Springer, Heidelberg (2005)
6. Backes, M., Pfitzmann, B., Waidner, M.: A Composable Cryptographic Library with Nested Operations. In: 10th ACM conference on Computer and Communications Security, pp. 220–230 (2003)
7. Bernstein, D.J.: SYN cookies (1996), `http://cr.yp.to/syncookies.html`
8. Cervesato, I., Durgin, N.A., Lincoln, P., Mitchell, J.C., Scedrov, A.: A meta-notation for protocol analysis. In: 12-th IEEE Computer Security Foundations Workshop, pp. 55–69 (1999)
9. Clavel, M., Durán, F., Eker, S., Meseguer, J., Lincoln, P., Martí-Oliet, N., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
10. Datta, A., Derek, A., Mitchell, J.C., Pavlovic, D.: A Derivation System and Compositional Logic for Security Protocols. J. Comput. Secur. 13(3), 423–482 (2005)
11. Denker, G., Meseguer, J., Talcott, C.: Protocol Specification and Analysis in Maude. In: Workshop on Formal Methods and Security Protocols (1998)
12. Denker, G., Meseguer, J., Talcott, C.: Rewriting Semantics of Meta-Objects and Composable Distributed Services. In: 3rd. Intl. Workshop on Rewriting Logic and its Applications (2000)
13. Durgin, N., Mitchell, J.C., Pavlovic, D.: A Compositional Logic for Proving Security Properties of Protocols. J. Comput. Secur. 11(4), 677–721 (2004)
14. Escobar, S., Meadows, C., Meseguer, J.: A Rewriting-Based Inference System for the NRL Protocol Analyzer: Grammar Generation. In: Formal Methods in Security Engineering, pp. 1–12 (2005)
15. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Berners-Lee, T.: RFC 2068: Hypertext Transfer Protocol – HTTP/1.1. Internet Society, (1997)
16. Goguen, J.A., Meseguer, J.: Order-sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations. Theoretical Computer Science 105(2), 217–273 (1992)
17. Gong, L., Syverson, P.: Fail-stop Protocols: An Approach to Designing Secure Protocols. In: 5th International Working Conference on Dependable Computing for Critical Applications, pp. 44–55 (1995)

18. Goodloe, A.E., Gunter, C.A.: Reasoning about Concurrency for Security Tunnels. In: IEEE Computer Security Foundations (CSF 2007) (2007)
19. Gunter, C.A., Khanna, S., Tan, K., Venkatesh, S.S.: DoS Protection for Reliably Authenticated Broadcast. In: Network and Distributed System Security Symposium (2004)
20. Gupta, A., Shmatikov, V.: Security Analysis of Voice-over-IP Protocols. Computer Security Foundations Symposium 00, 49–63 (2007)
21. Harkins, D., Carrel, D.: The Internet Key Exchange(IKE). Internet Society (1998)
22. He, C., Sundararajan, M., Datta, A., Derek, A., Mitchell, J.C.: A Modular Correctness Proof of IEEE 802.11i and TLS. In: 12th ACM conference on Computer and Communications Security, pp. 2–15 (2005)
23. Kaufman, C.: RFC 4306: Internet Key Exchange (IKEv2) Protocol. Internet Society (2005)
24. Khanna, S., Venkatesh, S.S., Fatemieh, O., Khan, F., Gunter, C.A.: Adaptive Selective Verification. In: IEEE Conference on Computer Communications (INFOCOM 2008) (2008)
25. Lafrance, S., Mullins, J.: An Information Flow Method to Detect Denial of Service Vulnerabilities. Journal of Unified Computer Science 9(11), 1350–1369 (2003)
26. Lichtenstein, O., Pnueli, A., Zuck, L.D.: The Glory of the Past. In: Conference on Logic of Programs, pp. 196–218 (1985)
27. Mahimkar, A., Shmatikov, V.: Game-based Analysis of Denial-of-Service Prevention Protocols. In: 18th IEEE workshop on Computer Security Foundations, pp. 287–301 (2005)
28. Maughan, D., Schertler, M.M., Schneider, Turner, J.: Internet Security Association and Key Management Protocol (ISAKMP). Internet Society (1998)
29. Meadows, C.: A Formal Framework and Evaluation Method for Network Denial of Service. In: 12th IEEE workshop on Computer Security Foundations (1999)
30. Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. Theor. Comput. Sci. 96(1), 73–155 (1992)
31. Meseguer, J., Talcott, C.: Semantic Models for Distributed Object Reflection. In: 16th European Conference on Object-Oriented Programming, pp. 1–36. Springer, Heidelberg (2002)
32. Mitchell, J., Durgin, N., Lincoln, P., Scedrov, A.: Multiset Rewriting and the Complexity of Bounded Security Protocols. Journal of Computer Security 12(2), 247–311 (2004)
33. Piper, D.: The Internet IP Security Domain Of Interpretation for ISAKMP. Internet Society (1998)
34. Schuba, C.L., Krsul, I.V., Kuhn, M.G., Spafford, E.H., Sundaram, A., Zamboni, D.: Analysis of a Denial of Service Attack on TCP. In: IEEE Symposium on Security and Privacy, pp. 208–223 (1997)
35. Sistla, A.P.: Safety, Liveness and Fairness in Temporal Logic. Formal Asp. Comput. 6(5), 495–512 (1994)
36. Yu, C.-F., Gligor, V.D.: A Specification and Verification Method for Preventing Denial of Service. IEEE Transactions on Software Engineering 16(6), 581–592 (1990)

Behavioural Theory at Work: Program Transformations in a Service-Centred Calculus*

Luís Cruz-Filipe[2], Ivan Lanese[1], Francisco Martins[2], António Ravara[3], and Vasco T. Vasconcelos[2]

[1] Computer Science Department, University of Bologna, Italy
[2] Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal
[3] Security and Quantum Information Group, Instituto de Telecomunicações, and Department of Mathematics, IST, Technical University of Lisbon, Portugal

Abstract. We analyse the relationship between object-oriented modelling and session-based, service-oriented modelling, starting from a typical UML Sequence Diagram and providing a program transformation into a service-oriented model. We also provide a similar transformation from session-based specifications into request-response specifications. All transformations are specified in SSCC—a process calculus for modelling and analysing service-oriented systems—and proved correct with respect to a suitable form of behavioural equivalence (full weak bisimilarity). Since the equivalence is proved to be compositional, results remain valid in arbitrary contexts.

1 Introduction

Web Services include a set of technologies to deploy applications over the Internet, making them dynamically searchable and composable, allowing great adaptability and reusability. While nowadays Web Services are one of the main technologies for implementing coordinated behaviours over the web, most of the modelling techniques commonly used are still tailored to the object-oriented development style, and only recently did dedicated development methodologies start emerging. For instance, extensions of UML [14] for Web Services are under development [21], and dedicated languages such as WS-BPEL [2] are increasingly popular.

As a contribution to bridge the gap between traditional object-oriented and emerging service-oriented models, and to allow the reuse of existing tools and techniques, we detail a model transformation procedure from a common object-oriented communication pattern into a session-based, service-oriented one. *Sessions* are a paradigm of service interaction that isolates the communications between two remote partners, making them safe from interferences. More complex orchestration scenarios are obtained by combining remote session communications and local communications. The expressive power of sessions has been shown, *e.g.,* in [3, 4, 8, 11, 12, 19, 20].

* This work has been partially sponsored by the project SENSORIA, IST–2005–016004, and by FCT, through the Multiannual Funding Programme and project SpaceTimeTypes, POSC/EIA/55582/2004.

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 59–77, 2008.

The work presented in this paper builds on top of SSCC [10, 12], a process calculus based on session communication and featuring service definition and service invocation as first class constructs. SSCC has been developed within the EU project Sensoria [18], by adapting previous proposals of calculi for services (such as [3, 6, 13]), while trying to find the best trade-off between expressiveness and suitability for formal analysis. SSCC provides dedicated constructs for service definition/invocation, session communication, and local communication based on streams, the later abstracting operating system local communication facilities.

We illustrate herein how UML Sequence Diagrams [1] (hereafter referred to as SDs) can be straightforwardly implemented in SSCC by exploiting suitable macros. While these macros hide auxiliary local services, making use of services for local communications is usually not tremendously efficient. We thus show how to transform these models into models where local communications are realised by streams, resulting in systems with a more session-oriented flavour. However, while sessions have proved quite useful for system modelling, current Web Service technologies (*e.g.,* WSDL [9] and WS-BPEL [2]) do not fully support them. In fact, complex interactions are usually based on the simpler request and request-response patterns. To bridge this mismatch we show how to break sessions into request-response sequences, easily implementable using current technologies.

An important feature of our work is that all transformations are proved correct with respect to the observational semantics of services. In fact, SSCC is equipped with a rigorous operational semantics, and with a behavioural theory based on an equivalence capturing the main aspects of service interaction. We show that our transformations preserve behavioural equivalence. Since we also prove that this equivalence—weak full bisimilarity over an early labelled transition system—is a congruence, the behaviour of every composition built exploiting these services remains unchanged when moving from the original programs into their transformed versions.

To the best of our knowledge, there are very few studies on weak bisimulations for service calculi other than SSCC. Bruni et al. present weak bisimulation for a calculus for multiparty sessions in SOC [5], and they apply it to show that an implementation of a service is compliant to a more abstract specification. However they do not provide a general theory or methodology for the transformation as we do. Vieira et al. discuss strong bisimulation in the realm of the Conversation Calculus [20], a variant of SCC [3]. They present a congruence result and axioms that express the spatial nature of their calculus and allow them to prove a normal form lemma. The behavioural theory, however, is not used in an application, as we do in this paper. Busi et al. study a bisimulation like relation to check conformance between a calculus for orchestration and a calculus for choreography [7], while we use it to check the correctness of transformations inside the same calculus. Most of the papers on program transformations (see, *e.g.,* [15] for a functional language) use program transformations to improve efficiency, whereas we are more interested in changing the style of the program, while making it implementable using current technologies. In the context of typed π-calculus, Davide Sangiorgi uses typed behavioural equivalences to prove a program transformation to increase concurrency in a system of objects [16].

The outline of the paper is as follows. The next section introduces SSCC. Then Section 3 discusses SDs and the corresponding SSCC code. Sections 4 and 5 study the bisimilarity relation, the former for general processes and the latter concentrating on sequential sessions. They pave the way for proving, in Section 6, the correctness of the transformations set forward in Section 3. Finally, Section 7 concludes the paper, pointing directions for further work.

2 SSCC

The Stream-based Session-Centred Calculus [10, 12] builds on its predecessor SCC [3], by introducing the ability to model complex orchestration patterns via a dedicated stream construct. Here we recall its syntax and its operational semantics.

Syntax. Processes use three kinds of identifiers: *service names* ranged over by a, b, x, y, ..., *stream names* ranged over by f, g, ..., and *process variables* ranged over by X, Y, ... The grammar in Figure 1 defines the *syntax of* SSCC *processes*.

P, Q	$::=$	*Processes*			
	$P \mid Q$	Parallel composition	$\mid$ $(\nu\, a)P$	Name restriction
$\mid$	$\mathbf{0}$	Terminated process	$\mid$ X	Process variable
$\mid$	$\mathsf{rec}\, X.P$	Recursive process definition	$\mid$ $a \Rightarrow P$	Service definition
$\mid$	$a \Leftarrow P$	Service invocation	$\mid$ $v.P$	Value sending
$\mid$	$(x)P$	Value reception	$\mid$ $\mathsf{stream}\, P\, \mathsf{as}\, f\, \mathsf{in}\, Q$	Stream
$\mid$	$\mathsf{feed}\, v.P$	Feed the process' stream	$\mid$ $f(x).P$	Read from a stream
v, w	$::=$	*Values*		
	a	Service name	$\mid$ unit	Unit value

Fig. 1. Syntax of SSCC

The first five cases introduce standard process calculi operators: parallel composition, restriction (for service names only), the terminated process, and recursion (we assume recursion guarded by prefixes). We then have two constructs to *build services*: definition or provider ($a \Rightarrow P$) and invocation or client ($a \Leftarrow P$). Both are defined by their name a and protocol P. Service definition and service invocation are symmetric. *Service protocols* are built using value sending ($v.P$) and receiving ($(x)P$), allowing bidirectional communication between clients and servers. Finally, there are three constructs for *service orchestration*. The stream construct declares a stream f for communication from P to Q. Process P can insert a value v into the stream using $\mathsf{feed}\, v.P'$, and process Q can read from stream f using $f(x).Q'$. Notice that stream names cannot be communicated, thus they model static channels. The rule of thumb concerning communication is the following: service interaction is intended for remote communication; stream interaction is intended for local communication.

Processes at run-time exploit an extended syntax: the interaction of a service definition and a service invocation produces an active session with two endpoints, one at the provider side and the other at the client side. Also, values in the stream are stored

P, Q	$::=$	$\ldots$ as in Figure 1	*Runtime processes*		
	$\mid$	$r \triangleright P$ Server session	$\mid$	$r \triangleleft P$	Client session
	$\mid$	$(\nu\, r)P$ Session restriction	$\mid$	$\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{v}\, \mathsf{in}\, Q$	Stream with values

Fig. 2. Run-time syntax of SSCC

together with the stream definition, *i.e.,* the stream has buffering capabilities. Let $\vec{v}$ (and sometimes $\vec{w}$) denote a (possibly empty) sequence of values. We introduce a fourth kind of identifier: *session names*, ranged over by $r, s, \ldots$, and use $n, m, \ldots$ to range over both session and service names. The grammar in Figure 2 defines the *syntax of run-time processes*. The constructor $\mathsf{stream}\, P\, \mathsf{as}\, f\, \mathsf{in}\, Q$ in Figure 1 is an abbreviation for its runtime version $\mathsf{stream}\, P\, \mathsf{as}\, f = \langle\rangle\, \mathsf{in}\, Q$.

In order to define the operational semantics we introduce a few auxiliary notations. We use $r \bowtie P$ to denote one of $r \triangleleft P$ and $r \triangleright P$, and we assume that when multiple $\bowtie$ appear in the same rule they are instantiated in the same direction ($\triangleleft$ or $\triangleright$), and that if $\overline{\bowtie}$ also appears, then it denotes the opposite direction.

Streams are considered ordered, acting as queues. We write $w :: \vec{v}$ for the stream obtained by enqueuing w to $\vec{v}$, and $\vec{v} :: w$ for a stream from which w can be dequeued. In the latter case $\vec{v}$ is what remains after dequeuing w.

Operational Semantics. Let $\mathrm{Set}(\vec{w}) = \{w_1, \ldots, w_n\}$, when $\vec{w} = w_1 \cdots w_n$ $(n \geq 0)$. As for bindings, name x is bound in $(x)P$ and in $f(x).P$; name n is bound in $(\nu\, n)P$; stream f is bound in $\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{v}\, \mathsf{in}\, Q$ with scope Q; and process variable X is bound in $\mathsf{rec}\, X.P$. Notation $\mathrm{fn}(P)$ (respectively $\mathrm{bn}(P)$) denotes the set of free (respectively bound) names in P, and $\mathrm{n}(P) = \mathrm{fn}(P) \cup \mathrm{bn}(P)$. We require processes to have no free process variables. The semantics of SSCC is defined using a labelled transition system (LTS, for short) in the early style.

Definition 1 (Labelled Transition System). *The rules in Figure 3, together with the symmetric version of rules* L-PAR*,* L-PAR-CLOSE*, and* L-SESS-COM-CLOSE*, inductively define the LTS on processes.*

In the LTS we use μ as a metavariable for labels, and extend $\mathrm{fn}(-)$, $\mathrm{bn}(-)$, and $\mathrm{n}(-)$ to labels. The only bound names in labels are r in service definition activation and service invocation and a in extrusion labels (conventionally, they are all in parenthesis). Label $\uparrow v$ denotes sending (the output) value v. Dually, label $\downarrow v$ is receiving (the input) value v. We use $\updownarrow v$ to denote one of $\uparrow v$ or $\downarrow v$, and we assume that when multiple $\updownarrow v$ appear in the same rule they are instantiated in the same direction, and that $\overline{\updownarrow} v$ denotes the opposite direction.

Continuing with the labels, $a \Leftarrow (r)$ and $a \Rightarrow (r)$ denote respectively the request and the activation of a service, where a is the name of the service, and r is the name of the new session to be created. We use $a \Leftrightarrow (r)$ to denote one of $a \Leftarrow (r)$ or $a \Rightarrow (r)$, and we assume that when multiple $a \Leftrightarrow (r)$ appear in the same rule they are instantiated in the same direction, and that $a \overline{\Leftrightarrow} (r)$ denotes the opposite direction. Furthermore, label $\Uparrow v$ denotes the feeding of value v into a stream, while label $f \Downarrow v$ reads value v from stream f. When an input or an output label crosses a session construct (rule L-SESS-VAL), we

$$v.P \xrightarrow{\uparrow v} P \qquad (x)P \xrightarrow{\downarrow v} P[v/x] \qquad \mathsf{feed}\, v.P \xrightarrow{\Uparrow v} P \qquad f(x).P \xrightarrow{f \Downarrow v} P[v/x]$$

(L-SEND, L-RECEIVE, L-FEED, L-READ)

$$\frac{P \xrightarrow{\mu} P' \quad \mu \neq \Uparrow v \quad \mathrm{bn}(\mu) \cap (\mathrm{fn}(Q) \cup \mathrm{Set}(\vec{w})) = \emptyset}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{\mu} \mathsf{stream}\, P'\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q} \quad \text{(L-STREAM-PASS-P)}$$

$$\frac{Q \xrightarrow{\mu} Q' \quad \mu \neq f \Downarrow v \quad \mathrm{bn}(\mu) \cap (\mathrm{fn}(P) \cup \mathrm{Set}(\vec{w})) = \emptyset}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{\mu} \mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'} \quad \text{(L-STREAM-PASS-Q)}$$

$$\frac{P \xrightarrow{\Uparrow v} P'}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{\tau} \mathsf{stream}\, P'\, \mathsf{as}\, f = v :: \vec{w}\, \mathsf{in}\, Q} \quad \text{(L-STREAM-FEED)}$$

$$\frac{Q \xrightarrow{f \Downarrow v} Q'}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w} :: v\, \mathsf{in}\, Q \xrightarrow{\tau} \mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'} \quad \text{(L-STREAM-CONS)}$$

$$\frac{r \notin \mathrm{fn}(P)}{a \Leftarrow P \xrightarrow{a \Leftarrow (r)} r \lhd P} \qquad \frac{r \notin \mathrm{fn}(P)}{a \Rightarrow P \xrightarrow{a \Rightarrow (r)} r \rhd P} \quad \text{(L-CALL, L-DEF)}$$

$$\frac{P \xrightarrow{\mu} P' \quad \mathrm{bn}(\mu) \cap \mathrm{fn}(Q) = \emptyset}{P|Q \xrightarrow{\mu} P'|Q} \qquad \frac{P \xrightarrow{\updownarrow v} P'}{r \bowtie P \xrightarrow{r \bowtie \updownarrow v} r \bowtie P'} \quad \text{(L-PAR, L-SESS-VAL)}$$

$$\frac{P[\mathsf{rec}\, X.P/X] \xrightarrow{\mu} P'}{\mathsf{rec}\, X.P \xrightarrow{\mu} P'} \qquad \frac{P \xrightarrow{r \bowtie \uparrow v} P' \quad Q \xrightarrow{r \boxtimes \downarrow v} Q'}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{r\tau} \mathsf{stream}\, P'\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'} \quad \text{(L-REC, L-SESS-COM-STREAM)}$$

$$\frac{P \xrightarrow{a \Leftrightarrow (r)} P' \quad Q \xrightarrow{a \Rightleftarrows (r)} Q'}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{\tau} (\nu\, r)\mathsf{stream}\, P'\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'} \quad \text{(L-SERV-COM-STREAM)}$$

$$\frac{P \xrightarrow{r \bowtie \uparrow v} P' \quad Q \xrightarrow{r \boxtimes \downarrow v} Q'}{P|Q \xrightarrow{r\tau} P'|Q'} \qquad \frac{P \xrightarrow{a \Leftrightarrow (r)} P' \quad Q \xrightarrow{a \Rightleftarrows (r)} Q'}{P|Q \xrightarrow{\tau} (\nu\, r)(P'|Q')} \quad \text{(L-SESS-COM-PAR, L-SERV-COM-PAR)}$$

$$\frac{P \xrightarrow{\mu} P' \quad n \notin \mathrm{n}(\mu)}{(\nu\, n)P \xrightarrow{\mu} (\nu\, n)P'} \qquad \frac{P \xrightarrow{r\tau} P'}{(\nu\, r)P \xrightarrow{\tau} (\nu\, r)P'} \qquad \frac{P \xrightarrow{\mu} P' \quad \mu \neq \updownarrow v \quad r \notin \mathrm{bn}(\mu)}{r \bowtie P \xrightarrow{\mu} r \bowtie P'} \quad \text{(L-RES,L-SESS-RES, L-SESS-PASS)}$$

$$\frac{P \xrightarrow{r \bowtie (v) \uparrow v} P' \quad Q \xrightarrow{r \boxtimes \downarrow v} Q' \quad v \notin \mathrm{fn}(Q)}{P|Q \xrightarrow{r\tau} (\nu\, v)(P'|Q')} \qquad \frac{P \xrightarrow{\mu} P' \quad \mu \in \{\uparrow a, r \bowtie \uparrow a, \Uparrow a\}}{(\nu\, a)P \xrightarrow{(a)\mu} P'} \quad \text{(L-PAR-CLOSE, L-EXTR)}$$

$$\frac{P \xrightarrow{r \bowtie (v) \uparrow v} P' \quad Q \xrightarrow{r \boxtimes \downarrow v} Q' \quad v \notin \mathrm{fn}(Q) \cup \mathrm{Set}(\vec{w})}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{r\tau} (\nu\, v)\mathsf{stream}\, P'\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q'} \quad \text{(L-SESS-COM-CLOSE)}$$

$$\frac{P \xrightarrow{(v) \Uparrow v} P' \quad v \notin \mathrm{fn}(Q) \cup \mathrm{Set}(\vec{w})}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{w}\, \mathsf{in}\, Q \xrightarrow{\tau} (\nu\, v)\mathsf{stream}\, P'\, \mathsf{as}\, f = v :: \vec{w}\, \mathsf{in}\, Q} \quad \text{(L-FEED-CLOSE)}$$

Fig. 3. Labelled Transition System

$$P|\mathbf{0} \equiv P \qquad P|Q \equiv Q|P \qquad (P|Q)|R \equiv P|(Q|R) \qquad \text{(S-NIL, S-COMM, S-ASSOC)}$$

$$(\nu\, n)P|Q \equiv (\nu\, n)(P|Q) \quad \text{if } n \notin \mathrm{fn}(Q) \qquad r \bowtie (\nu\, a)P \equiv (\nu\, a)(r \bowtie P) \qquad \text{(S-EXTR-PAR, S-EXTR-SESS)}$$

$$\mathsf{stream}\,(\nu\, a)P\,\mathsf{as}\,f = \vec{v}\,\mathsf{in}\,Q \equiv (\nu\, a)(\mathsf{stream}\,P\,\mathsf{as}\,f = \vec{v}\,\mathsf{in}\,Q) \quad \text{if } a \notin \mathrm{fn}(Q) \cup \mathrm{Set}(\vec{v}) \qquad \text{(S-EXTR-STREAML)}$$

$$\mathsf{stream}\,P\,\mathsf{as}\,f = \vec{v}\,\mathsf{in}\,(\nu\, a)Q \equiv (\nu\, a)(\mathsf{stream}\,P\,\mathsf{as}\,f = \vec{v}\,\mathsf{in}\,Q) \quad \text{if } a \notin \mathrm{fn}(P) \cup \mathrm{Set}(\vec{v}) \qquad \text{(S-EXTR-STREAMR)}$$

$$(\nu\, n)(\nu\, m)P \equiv (\nu\, m)(\nu\, n)P \qquad (\nu\, a)\mathbf{0} \equiv \mathbf{0} \qquad \mathsf{rec}\,X.P \equiv P[\mathsf{rec}\,X.P/X] \qquad \text{(S-SWAP, S-COLLECT, S-REC)}$$

Fig. 4. Structural congruence

add to the label the name of the session and whether it is a server or a client session (for example, $\downarrow v$ may become $r \lhd \downarrow v$).

The label denoting a conversation step in a free session r is $r\tau$, and if the value passed in the session channel is private, it remains private in the resulting process; rules L-PAR-CLOSE and L-SESS-COM-CLOSE accomplish this using the usual π-calculus approach. A label τ is obtained only when r is restricted (rule L-SESS-RES). Thus a τ action can be obtained in four cases: a communication inside a restricted session, a service invocation, a feed or a read from a stream. Notice also that we have two contexts causing interaction: parallel composition and stream. Finally, bound actions, $(a)\mu$, represent the extrusion of a in their respective free counterparts μ.

This LTS is slightly different from its original version [12]. In particular it does not exploit structural congruence, in order to simplify some proofs. The two LTSs are, however, equivalent up to the structural congruence relation (inductively defined by the rules in Figure 4). Since we show in Lemma 2 that structural congruence is included in strong full bisimilarity, all our results are valid also for the original LTS. A detailed equivalence proof can be found in a technical report [10].

Some processes, such as $r \lhd r \lhd P$, can be written using the run-time syntax, but they are not reachable from processes in the basic syntax of Figure 1. We consider these processes ill-formed, and therefore make the following assumption, which is necessary for most results.

Assumption 1. *Henceforth, all processes under consideration are either in the syntax of Figure 1, or can be obtained from these via LTS transitions.*

3 Common Sequence Diagram Patterns

UML Sequence Diagrams [1] describe the exchange of messages among the components in a complex system. We present a typical SD, and then describe how the same behaviour can be modelled first in a more session-centred style, and then in a request-response style suitable for implementation. Diagrams for the initial SD and those describing the result of each transformation are in Figures 5 to 7.

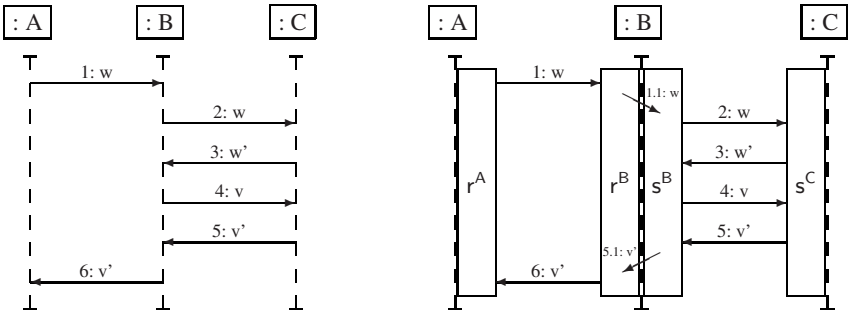


Fig. 5. Sequence diagram communication pattern: object-centred and session-centred view

Object-Oriented View. The SD on the left of Figure 5 describes a very common pattern appearing in scenarios involving (at least) three partners. The description of the communication pattern is as follows.

> Object B receives from object A the value w and forwards it to object C. After receiving the value, object C answers with a value w'. Object B replies with v and finally object C replies with v'. Now object B forwards it to object A.

Notice that by "Object B receives from object A the value w" we mean that object A invokes a method in object B passing the value w.

Session-Centred View. Assume that the components of this abstract communication scenario are clients and servers of a service-oriented architecture, and further assume that communication happens via sessions. We refine the diagram by incorporating information about the running sessions, in the diagram on the right of Figure 5, where the slanted arrows mean message passing between sessions[1]. An instance of service B (let us call these instances *participants*) has a session r running with an instance of service A and another session s running with an instance of service C. Since sessions involve two partners, a session r between instances of services A and B has two sides—called endpoints, r^A at the instance of service A and r^B at the instance of service B. The communication pattern is now like this:

> Participant B receives in session r^B the value w, passes it to its part of the session with participant C (s^B), and then forwards the value through this session to C. Inside the same session C sends w' to B, B sends v to C and C sends v' to B. Participant B now forwards the value v' back to A, passing it from session s to session r.

In addition to the normal constructs in the calculus, to model object-oriented systems (that do not follow the laws of session communication), it is useful to have two

[1] Since to the best of our knowledge no extension of SDs with session information exists, we introduce a notation for it.

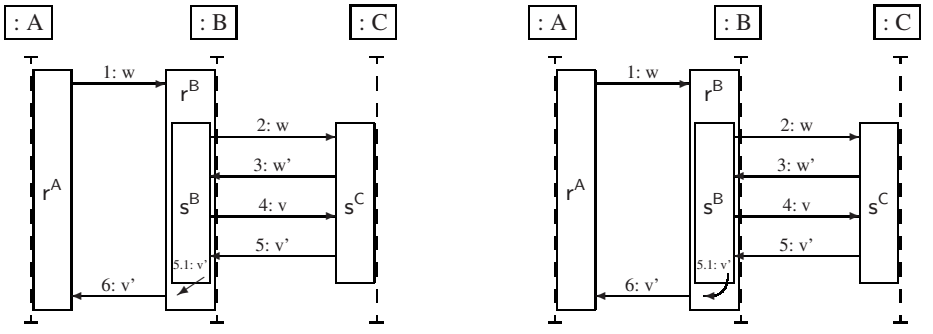


Fig. 6. Sequence diagram: subsession communicating via message passing and via a stream

constructs enabling arbitrary message passing. These can be expressed in SSCC by exploiting fresh auxiliary services.

$$b \Uparrow \langle v_1, ..., v_n \rangle.P \triangleq \mathsf{stream}\, b \Leftarrow v_1 ... v_n.\mathsf{feed\ unit\ as}\ f\ \mathsf{in}\ f(v).P$$

$$b \Downarrow (x_1, ..., x_n)P \triangleq \mathsf{stream}\, b \Rightarrow (z_1)...(z_n).\mathsf{feed}\, z_1 ... \mathsf{feed}\, z_n\ \mathsf{as}\ f\ \mathsf{in}\ f(x_1)...f(x_n).P$$

where name v and stream f are not used in P.

The diagram on the right of Figure 5 is directly implemented in SSCC as

$$\mathsf{SC} \triangleq (\nu\, b, c)\ (\ \mathsf{A} \mid \mathsf{B} \mid \mathsf{C}\ ),$$

where

$$\mathsf{A} \triangleq b \Leftarrow w.(y)P,\ \mathsf{B} \triangleq (\nu\, b_1, b_2)\ (\ \mathsf{B}_1 \mid \mathsf{B}_2\ ),\ \text{and}\ \mathsf{C} \triangleq c \Rightarrow (x)w'.(y)v'.S,$$

with

$$\mathsf{B}_1 \triangleq b \Rightarrow (x)b_1 \Uparrow x.b_2 \Downarrow (y)y.Q,\ \text{and}\ \mathsf{B}_2 \triangleq c \Leftarrow b_1 \Downarrow (x)x.(z)v.(y)b_2 \Uparrow y.R.$$

The process above, although not deterministic (its first step may either be the invocation of service b or of service c), is confluent, and it is easy to check that its behaviour reflects the one described on the right of Figure 5.

A First Optimisation. When the participant B has the value sent by A, it may immediately send it to participant C, calling it (and thus opening a subsession). One simply has to perform a "local" transformation on B. The resulting diagram is on the left of Figure 6, and it is implemented in SSCC as process SC′, where we denote by E the new instance of B.

$$\mathsf{SC}' \triangleq (\nu\, b, c)\ (\ \mathsf{A} \mid \mathsf{E} \mid \mathsf{C}\ ) \qquad \text{where}$$

$$\mathsf{E} \triangleq b \Rightarrow (x)(\nu\, b_1)(c \Leftarrow x.(z)v.(y)b_1 \Uparrow y.R \mid b_1 \Downarrow (y)y.Q)$$

Still, E (which replaces B from the previous version) needs to pass the value sent by C in their session, to its session with A, and a communication based on an auxiliary service is used. Notice that the process SC′ is deterministic.

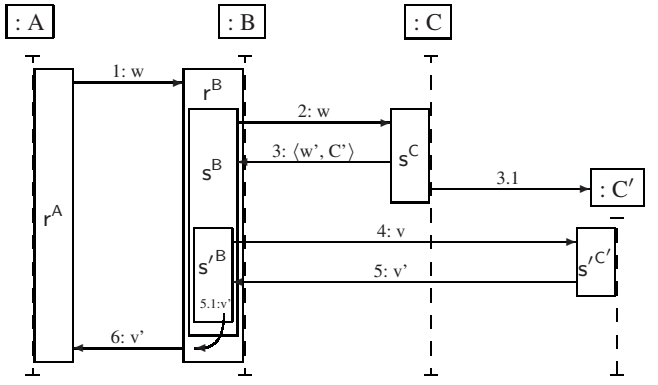


Fig. 7. Sequence diagram communication pattern: using subsessions and continuations

A Second Optimisation. The transfer of a value from a subsession to its parent session is, in the previous implementation, not straightforward, since it requires the use of local services. In fact, as discussed in [12], it is more convenient to use a trans-session construct like stream. Now participant F (the initial B), passes the value received from C from the subsession to the main session using another communication construct—a stream—instead of using a local service. This is implemented in process SC″, also deterministic.

$$\mathrm{SC}'' \triangleq (\nu\, b, c)\ (\ \mathsf{A} \mid \mathsf{F} \mid \mathsf{C}\ )$$

where the new code for B (now F) is as follows.

$$\mathsf{F} \triangleq b \Rightarrow (x)(\mathsf{stream}\ c \Leftarrow x.(z)v.(y)\mathsf{feed}\ y.R\ \mathsf{as}\ f\ \mathsf{in}\ f(y).y.Q)$$

The corresponding diagram is on the right of Figure 6. The two diagrams in Figure 6 are quite similar: the one on the left uses message passing (denoted by the straight arrow from the inner to the outer session), whereas the one on the right uses stream communication (described by the curved arrow).

Implementing the Diagram. The previous diagram, and the corresponding SSCC process, model the pattern at hand in a service-oriented session-based style. However, current Web Service technologies do not provide support for a complex mechanism such as sessions, considering instead request (one-way communication) and request-response (two one-way communications in opposite directions) only—see, *e.g.,* the definition of WSDL [9]. It is easy to see that these are particular cases of sessions, where protocols are composed respectively by one output or by one output followed by one input. The new communication pattern is described in Figure 7, and reads as follows.

> Participant B receives in session r^{B} the value w, and then forwards it through its session with participant C (s^{B}) to C itself. Inside the same session C sends to B value w' together with the name of a freshly generated service C' to continue the conversation on. Now B invokes C' creating a new subsession s' of session s and, inside s', B sends v and receives as answer v'. Participant B now forwards the value v' back to A, passing it from session s' to session r.

This pattern can be implemented as:

$$\mathsf{SC}''' \triangleq (\nu\, b, c)\ (\ \mathsf{A} \mid \mathsf{G} \mid \mathsf{D}\ )$$

where the new codes for B (now G) and C (now D) are below. To write these new codes we need to consider polyadic inputs and outputs, denoted respectively by $(x_1, \ldots, x_n)$ and $\langle v_1, \ldots, v_n \rangle$. They can be easily accommodated in the theory.

$$\mathsf{G} \triangleq b \Rightarrow (x)(\mathsf{stream}\ c \Leftarrow x.(z, c')c' \Leftarrow v.(y)\mathsf{feed}\, y.R\ \mathsf{as}\ f\ \mathsf{in}\ f(y).y.Q)$$

$$\mathsf{D} \triangleq c \Rightarrow (x)(\nu\, c')\langle w', c' \rangle.c' \Rightarrow (y)v'.S$$

Naturally, one asks whether the transformations of SC into SC′, SC″ and finally SC‴ are correct, not changing the observable behaviour of processes. The next sections introduce suitable tools to give a positive answer to this question in Section 6.

4 Bisimilarity in SSCC

We study the usual two notions of bisimilarity, strong and weak. Both are non-input congruences in the class of SSCC processes. One can get a congruence by considering (strong or weak) full bisimilarity, *i.e.,* by closing bisimilarity with respect to service name substitutions (notice that there is no reason to close with respect to session or stream names, since no substitutions are performed on them). Although the general strategy is the same as for the π-calculus, the proof techniques themselves differ significantly. Herein we only present the main results. Detailed proofs can be found in a technical report [10].

Section 2 defines a labelled transition system (LTS) in the early style; we now study the notions of bisimilarity known in the literature as *ground* and *full*. The reason for choosing these is simple: we are interested in a compositional equivalence, *i.e.,* equivalent services should behave the same even if used as parts of complex systems. Therefore, we choose the simplest possible setting where this may happen. It is well-known already from the π-calculus that ground bisimilarity over a late LTS is not preserved by parallel composition, requiring the more demanding notions of late and early bisimilarity (which in turn are not preserved by input prefix, since they are not closed under general substitutions). Not surprisingly, this fact also occurs in SSCC: ground bisimilarity is a non-input congruence. Sangiorgi and Walker present counter-examples for the preservation of both strong and weak bisimilarities in the synchronous π-calculus without sum and match [17, pp. 224–225]. Of these, the first can be easily translated to our language, using services to mimic π's input and output constructors; for the weak case, the counter-example is not so directly transposable, but it can still be adapted.

Strong Bisimilarity. The relation, hereafter referred to simply as "bisimilarity", is defined as usual over the class of all processes.

Definition 2 (Strong Bisimilarity). *A symmetric binary relation $\mathcal{R}$ on processes is a* (strong) bisimulation *if, for any processes P, Q such that $P\mathcal{R}Q$, if $P \xrightarrow{\alpha} P'$ with $\mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$, then there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $P'\ \mathcal{R}\ Q'$. (Strong)*

bisimilarity $\sim$ is the largest bisimulation. Two processes P and Q are (strong) bisimilar *if $P \sim Q$.*

Also, a full bisimulation is a bisimulation closed under service name substitutions, and we call full bisimilarity $\sim_{\mathrm{f}}$ the largest full bisimulation.

Since bisimilarity is not closed under service name substitutions, $\sim_{\mathrm{f}} \subsetneq \sim$.

Notice that bisimilarity (respectively full bisimilarity) can be obtained as the union of all bisimulations (respectively full bisimulations) or as a fixed-point of a suitable monotonic operator; moreover, as expected, structurally congruent processes (Figure 4) are bisimilar.

Lemma 1 (Harmony Lemma). *Let P and Q be processes with $P \equiv Q$. If $P \xrightarrow{\alpha} P'$, then $Q \xrightarrow{\alpha} Q'$ with $P' \equiv Q'$, and vice-versa.*

Lemma 2. *Structurally congruent processes are full bisimilar.*

As in the π-calculus, bisimilarity is a non-input congruence, *i.e.,* it is closed under contexts different from value reception and input from stream.

Theorem 1. *Bisimilarity is a non-input congruence.*

Corollary 1. *Full bisimilarity is a congruence.*

Weak Bisimilarity. As usual, we introduce some abbreviations: $P \stackrel{\tau}{\Rightarrow} Q$ iff $P \xrightarrow{\tau} \cdots \xrightarrow{\tau} Q$ and $P \stackrel{\alpha}{\Rightarrow} Q$ iff $P \stackrel{\tau}{\Rightarrow} \xrightarrow{\alpha} \stackrel{\tau}{\Rightarrow} Q$ for $\alpha \neq \tau$. In particular, $P \stackrel{\tau}{\Rightarrow} P$ for every process P.

Definition 3. *A symmetric binary relation $\mathcal{R}$ on processes is a* weak bisimulation *if, for any processes P, Q such that $P \mathcal{R} Q$, if $P \stackrel{\alpha}{\Rightarrow} P'$ with $\mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \emptyset$, then there exists a process Q' such that $Q \stackrel{\alpha}{\Rightarrow} Q'$ and $P' \mathcal{R} Q'$. Weak bisimilarity $\approx$ is the largest weak bisimulation. Two processes P and Q are said to be* weak bisimilar *if $P \approx Q$.*

Also, a full weak bisimulation is a weak bisimulation closed under service name substitutions, and we call full weak bisimilarity $\approx_{\mathrm{f}}$ the largest full weak bisimulation.

Again, weak bisimilarity can be obtained as the union of all weak bisimulations or as a fixed-point of a suitable monotonic operator. Moreover, as for the strong case, weak bisimilarity is a non-input congruence, as in in the π-calculus.

Theorem 2. *Weak bisimilarity is a non-input congruence.*

Corollary 2. *Weak full bisimilarity is a congruence.*

Useful Axioms. Even if presenting a complete axiomatization for such a complex calculus is out of the scope of this paper, we present here some axioms (equational laws correct with respect to strong/weak full bisimilarity) that capture key facts about the behaviour of processes. Some of them are useful to prove the correctness of the transformations presented in the previous section. We need the following definitions of contexts and double contexts.

Definition 4 (Contexts and double contexts). *A context $\mathcal{C}[\![\bullet]\!]$ is any* SSCC *process where a subterm has been replaced by the symbol $\bullet$. The application $\mathcal{C}[\![P]\!]$ of context $\mathcal{C}[\![\bullet]\!]$ to process P is the process obtained by replacing $\bullet$ with P inside $\mathcal{C}[\![\bullet]\!]$.*

A double context $\mathcal{D}[\![\bullet_1, \bullet_2]\!]$ is any SSCC *process where two subterms have been replaced by symbols $\bullet_1$ and $\bullet_2$. The application $\mathcal{D}[\![P_1, P_2]\!]$ of double context $\mathcal{D}[\![\bullet_1, \bullet_2]\!]$ to processes P_1 and P_2 is the process obtained by replacing $\bullet_1$ with P_1 and $\bullet_2$ with P_2 inside $\mathcal{D}[\![\bullet_1, \bullet_2]\!]$.*

Unless otherwise specified, the correctness of the axioms below can be proved by considering as full bisimulation all the instances of the equations together with the identity.

Proposition 1

Session Garbage Collection

$$(\nu\, \mathsf{r})\mathcal{D}[\![\mathsf{r} \rhd \mathbf{0}, \mathsf{r} \lhd \mathbf{0}]\!] \sim_{\mathrm{f}} \mathcal{D}[\![\mathbf{0}, \mathbf{0}]\!] \qquad \textit{where } \mathcal{D} \textit{ does not bind } r \tag{1}$$

Stream Garbage Collection

$$\mathsf{stream}\ \mathbf{0}\ \mathsf{as}\ f\ \mathsf{in}\ P \sim_{\mathrm{f}} P \qquad \textit{if } f \textit{ does not occur in } P \tag{2}$$

Session Independence

$$r \bowtie \mathsf{Q} \mid s \bowtie \mathsf{P} \sim_{\mathrm{f}} r \bowtie (s \bowtie \mathsf{Q} \mid \mathsf{P}) \qquad \textit{if } s \neq r \tag{3}$$

The same holds if the sessions have opposite polarities.

Stream Independence

$$\mathsf{stream}\ P\ \mathsf{as}\ f\ \mathsf{in}\ \mathsf{stream}\ P'\ \mathsf{as}\ g\ \mathsf{in}\ Q \sim_{\mathrm{f}} \mathsf{stream}\ P'\ \mathsf{as}\ g\ \mathsf{in}\ \mathsf{stream}\ P\ \mathsf{as}\ f\ \mathsf{in}\ Q \qquad \textit{if } f \neq g \tag{4}$$

Streams are Orthogonal to Sessions

$$r \bowtie (\mathsf{feed}\, v \mid P) \sim_{\mathrm{f}} \mathsf{feed}\, v \mid r \bowtie P \tag{5}$$

Stream Locality

$$\mathsf{stream}\ P\ \mathsf{as}\ f\ \mathsf{in}\ (Q \mid Q') \sim_{\mathrm{f}} (\mathsf{stream}\ P\ \mathsf{as}\ f\ \mathsf{in}\ Q) \mid Q', \quad \textit{if } f \notin \mathrm{fn}(Q') \tag{6}$$

Unused Stream

$$\mathsf{stream}\ P\ \mathsf{as}\ f\ \mathsf{in}\ 0 \approx_{\mathrm{f}} P\{\mathsf{feed}\, v.Q \to Q\} \tag{7}$$

Parallel Composition Versus Streams

$$\mathsf{stream}\ P\ \mathsf{as}\ f\ \mathsf{in}\ Q \sim_{\mathrm{f}} P|Q \qquad \textit{if } f \notin \mathrm{fn}(Q) \textit{ and } P \textit{ does not contain } \mathsf{feed} \tag{8}$$

The Session Independence law shows that different sessions are independent. Interestingly this property is strongly dependent on the available operators, and fails in similar calculi such as [3, 4].

The notation $\{\mathsf{feed}\, v.Q \to Q\}$ in the Unused Stream law (Axiom 7) denotes a transformation on processes defined by induction on the syntax which is the identity but for transforming $\mathsf{feed}\, v.Q$ into Q. Notice also that Axiom 7 is correct only with respect to full weak bisimilarity. It becomes correct with respect to strong full bisimilarity if and only if P does not contain any feed which is not inside another stream. Together with Equation 2 this allows to prove the relation between streams and parallel composition in Equation 8.

$$\frac{P : U}{v.P : !.U} \qquad \frac{P : U}{(x)P : ?.U} \qquad \frac{P : U}{a \Rightarrow P : \mathsf{end}} \qquad \frac{P : U}{a \Leftarrow P : \mathsf{end}}$$

(T-SEND, T-RECEIVE, T-DEF, T-CALL)

$$\frac{P : U}{r \triangleright P : \mathsf{end}} \qquad \frac{P : U}{r \triangleleft P : \mathsf{end}} \qquad \frac{P : U}{\mathsf{feed}\, v.P : U} \qquad \frac{P : U}{f(x).P : U}$$

(T-SESS-S, T-SESS-C, T-FEED, T-READ)

$$\frac{P : U \quad Q : \mathsf{end}}{P|Q : U} \qquad \frac{P : \mathsf{end} \quad Q : U}{P|Q : U}$$

(T-PAR-L, T-PAR-R)

$$\frac{P : U \quad Q : \mathsf{end}}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{v}\, \mathsf{in}\, Q : U} \qquad \frac{P : \mathsf{end} \quad Q : U}{\mathsf{stream}\, P\, \mathsf{as}\, f = \vec{v}\, \mathsf{in}\, Q : U}$$

(T-STREAM-L, T-STREAM-R)

$$\frac{P : \mathsf{end}}{\mathsf{rec}\, X.P : \mathsf{end}} \qquad \frac{P : U}{(\nu\, n)P : U} \qquad X : \mathsf{end} \qquad \mathbf{0} : \mathsf{end}$$

(T-REC, T-RES, T-VAR, T-NIL)

Fig. 8. Type system for sequentiality

5 Breaking Sequential Sessions

The equations shown in Section 4 hold for general processes, and will allow us to prove the correctness of the two optimizations in Section 3. However to prove the correctness of the implementation step they are not enough.

In fact, it is not easy to break a session allowing the conversation to continue in a freshly generated new session, since, in general, communication patterns inside sessions can be quite complex, *e.g.,* since sessions may include many ongoing concurrent communications. However, a small class of sequential sessions captures the most interesting within-session behaviours. Such a class can be identified by a type system.

We start by presenting the type system for sequentiality, and then we will present some properties of well-typed processes that will allow us to prove the correctness of the last transformation in Section 3. The type system is a simplification of the one in [12], which guarentees also protocol compatibility. Moreover, we consider just finite types, thus session protocols should be finite. Notice that this constraint does not forbid infinite behaviours, but just infinite sessions. In particular, if a process is typable according to the type system in [12], and all the involved types are non recursive, then the process is typable according to the type system presented herein.

We consider typed processes of the form $P : U$ where U is the protocol type. We consider as types $?.U$, $!.U$, and end, denoting respectively a protocol that performs an input and then continues as prescribed by U, a protocol that performs an output and then continues as prescribed by U, and the terminated protocol. It is clear that in this setting a request is a session with protocol $!.\mathsf{end}$ (and complementary protocol $?.\mathsf{end}$), while a request-response has protocol $!.?.\mathsf{end}$ (and complementary protocol $?.!.\mathsf{end}$). The type system is inductively defined by the rules in Figure 8.

Under the typability assumption, SSCC sessions are sequential in a very strong sense: we can statically define a correspondence between inputs and outputs such that each input is always matched by the corresponding output. We show how to break

sessions, *i.e.,* how to make the conversation continue on a freshly created new session. The general law is presented under Theorem 3. The two pieces of sessions have protocols that are simpler than the original one, thus by repeatedly applying the transformation we can reduce any protocol to a composition of request and request-response patterns. We formalise the procedure described so far.

Definition 5. *Let P be a process. An input/output prefix inside P is* at top-level in P *if it is neither inside a service definition/invocation nor inside a session. Given a process P we can assign sequential indices to top-level input/output prefixes in P according to the position of their occurrence in the term, starting from* 1*. Thus the i-th* top-level prefix in P *is the top-level prefix in P that occurs in i-th position.*

For instance, let P be $a.(x).\mathsf{stream}\,(y).\mathsf{feed}\,y\,\mathsf{as}\,f\,\mathsf{in}\,f(y).y \Leftarrow a.(z).\mathsf{feed}\,z$. Then P annotated with indices on its top-level prefixes is:

$$a : 1.(x) : 2.\mathsf{stream}\,(y) : 3.\mathsf{feed}\,y\,\mathsf{as}\,f\,\mathsf{in}\,f(y).y \Leftarrow a.(z).\mathsf{feed}\,z$$

Definition 6 (Active contexts). *Active contexts are contexts defined by the following grammar.*

$$\begin{aligned}\mathcal{A}[\![\bullet]\!] \;::=\; & \bullet \;\mid\; \mathcal{A}[\![\bullet]\!]|Q \;\mid\; P|\mathcal{A}[\![\bullet]\!] \;\mid\; (\nu\, n)\mathcal{A}[\![\bullet]\!] \;\mid\; r \bowtie \mathcal{A}[\![\bullet]\!] \\ & \mid\; \mathsf{stream}\,\mathcal{A}[\![\bullet]\!]\,\mathsf{as}\,f = \vec{v}\,\mathsf{in}\,Q \;\mid\; \mathsf{stream}\,P\,\mathsf{as}\,f = \vec{v}\,\mathsf{in}\,\mathcal{A}[\![\bullet]\!]\end{aligned}$$

Definition 7. *Given a process P with a subterm Q, we say that Q is enabled in P if $P = \mathcal{A}[\![Q]\!]$ for some active context $\mathcal{A}[\![\bullet]\!]$. The same definition holds also for prefixes.*

Intuitively a subterm is enabled when it can execute.

Lemma 3. *Let P be a typable process. If P has type* end *then it has no top-level enabled prefixes, otherwise it has exactly one top-level enabled prefix, and this has index* 1.

Proof. By induction on the typing proof of P. The thesis follows trivially for rules T-SEND, T-RECEIVE, T-DEF, T-CALL, T-SESS-S, T-SESS-C, T-FEED, T-READ, and T-NIL. For rule T-RES, it follows by inductive hypothesis. For rules T-PAR-L, T-PAR-R, T-STREAM-L, and T-STREAM-R, it follows from the observation that one of the two sides has no enabled prefix, thus inductive hypothesis can be applied on the other side.

Definition 8. *Given a transition $P \xrightarrow{\alpha} Q$ and a prefix in P we say that the prefix is consumed if, in the derivation of the transition, rule* L-SEND *or* L-RECEIVE *is applied to the prefix.*

Notice that the consumed prefix does not occur in Q.

Lemma 4. *Let $P_0 = (\nu\, a)\mathcal{D}[\![a \Rightarrow P, a \Leftarrow Q]\!]$ be a typed process such that a does not occur in $\mathcal{D}[\![\bullet, \bullet]\!]$. Suppose $P_0 \xrightarrow{\alpha_0} \ldots \xrightarrow{\alpha_{n-1}} P_n$. Then either:*

- *$P_n = (\nu\, a)\mathcal{D}'[\![a \Rightarrow P, a \Leftarrow Q]\!]$ for some $\mathcal{D}'[\![\bullet, \bullet]\!]$ or*

– $P_n = (\nu\, r)\mathcal{D}'[\![r \rhd P', r \lhd Q']\!]$ *for some* $\mathcal{D}'[\![\bullet, \bullet]\!]$*, processes* P' *and* Q' *and session name* r*.*

Furthermore, whenever the i*-th prefix in* P *is consumed the* i*-th prefix in* Q *is consumed too, and the two are synchronised.*

Proof. The proof is by induction on n. The base case ($n = 0$) is trivial. Let us consider the inductive case. In order to prove the second condition we show that the following property holds: at each step either (i) all the prefixes preserve their indices, or (ii) prefixes with index 1 are consumed and the indices of all other prefixes decrease by 1.

Let us consider the two possible forms for P_i. In both the cases if the transition involves only the context then the thesis follows trivially (case (i)). In the first case the only other possible transition is the interaction between the service invocation and the service definition in the two holes (since a does not occur in $\mathcal{D}[\![\bullet, \bullet]\!]$). This leads to a process of the second form. Also, the indices are preserved. Let us consider the second case. For transitions not involving prefixes the thesis follows trivially (case (i)). Suppose now that *e.g.,* in P' an output prefix is consumed (the case where the prefix is an input prefix or the consumed prefix is in Q' is symmetric). Thus $P' \xrightarrow{\uparrow v} P''$ and $r \rhd P' \xrightarrow{r \rhd \uparrow v} r \rhd P''$. Since r is private, this label should interact with a label of the form $r \lhd \downarrow v$. For the condition about well-formed processes this can be generated only by $r \lhd Q'$, and thanks to Lemma 3 this must be the prefix with index 1. Thus in both P'' and Q'' prefix indices are equal to the indices in P' and Q' minus 1. This proves the thesis.

We now have all the tools required to prove the correctness of the session breaking technique.

Theorem 3. *Let* $(\nu\, a)\mathcal{D}[\![a \Leftarrow \mathcal{C}[\![(x).P]\!], a \Rightarrow \mathcal{C}'[\![v.Q]\!]]\!]$ *be a typed process such that* a *does not occur in* $\mathcal{D}[\![\bullet, \bullet]\!]$*. Suppose that there exists* i *such that* $(x).P$ *and* $v.Q$ *are the* i*-th top-level prefixes in* $\mathcal{C}[\![(x).P]\!]$ *and in* $\mathcal{C}'[\![v.Q]\!]$*, respectively. Let* $y \notin \mathrm{fn}(P)$*,* $b \notin \mathrm{fn}(Q)$*. Then:*

$$(\nu\, a)\mathcal{D}[\![a \Leftarrow \mathcal{C}[\![(x).P]\!], a \Rightarrow \mathcal{C}'[\![v.Q]\!]]\!] \approx_{\mathrm{f}} (\nu\, a)\mathcal{D}[\![a \Leftarrow \mathcal{C}[\![(x, y).y \Leftarrow P]\!], a \Rightarrow \mathcal{C}'[\![(\nu\, b)\langle v, b\rangle.b \Rightarrow Q]\!]]\!]$$

Proof. We show that the following is a full weak bisimulation:

$$\begin{aligned}\{&((\nu\, a)\mathcal{D}[\![a \Leftarrow \mathcal{C}[\![(x).P]\!], a \Rightarrow \mathcal{C}'[\![v.Q]\!]]\!], \\ &\quad (\nu\, a)\mathcal{D}[\![a \Leftarrow \mathcal{C}[\![(x, y).y \Leftarrow P]\!], a \Rightarrow \mathcal{C}'[\![(\nu\, b)\langle v, b\rangle.b \Rightarrow Q]\!]]\!]) \\ &((\nu\, r)\mathcal{D}[\![r \lhd \mathcal{C}[\![(x).P]\!], r \rhd \mathcal{C}'[\![v.Q]\!]]\!], \\ &\quad (\nu\, r)\mathcal{D}[\![r \lhd \mathcal{C}[\![(x, y).y \Leftarrow P]\!], r \rhd \mathcal{C}'[\![(\nu\, b)\langle v, b\rangle.b \Rightarrow Q]\!]]\!]) \\ &((\nu\, r)\mathcal{D}[\![r \lhd \mathcal{C}[\![P]\!], r \rhd \mathcal{C}'[\![Q]\!]]\!], (\nu\, r, b)\mathcal{D}[\![r \lhd \mathcal{C}[\![b \Leftarrow P]\!], r \rhd \mathcal{C}'[\![b \Rightarrow Q]\!]]\!]) \\ &((\nu\, r)\mathcal{D}[\![r \lhd \mathcal{C}[\![P]\!], r \rhd \mathcal{C}'[\![Q]\!]]\!], (\nu\, r, r')\mathcal{D}[\![r \lhd \mathcal{C}[\![r' \lhd P]\!], r \rhd \mathcal{C}'[\![r' \rhd Q]\!]]\!])\}\end{aligned}$$

where all the names and processes are universally quantified, $y \notin \mathrm{fn}(P)$, $b \notin \mathrm{fn}(Q)$, and $(x).$ and $v.$ have the same index.

Processes in the first pair can move only to processes of the same shape or to processes in the second pair because of Lemma 4. Similarly processes in the second pair can move to processes of the same form or to processes in the third pair (notice in fact that prefixes with the same index should be consumed at the same time). In the third pair the only transition that can change the structure of the processes is the invocation of service b on the right, but since this is a τ step, thus the left part can answer by staying idle.

Notice that the technique above can be extended so to break protocols of services with more than one definition/invocation: simply break all of them at the same point.

6 Correctness of the Transformations

We now prove that the transformations presented in Section 3 are actually correct with respect to full weak bisimilarity. Interestingly, they are also transparent for process A, i.e. A needs not to be changed when the transformation is applied, and binder $(\nu\,\mathsf{b})$. To prove this we show that the three equations below hold.

$$(\nu\,\mathsf{c})(\mathsf{B} \mid \mathsf{C}) \approx_{\mathrm{f}} (\nu\,\mathsf{c})(\mathsf{E} \mid \mathsf{C}) \tag{9}$$

$$(\nu\,\mathsf{c})(\mathsf{E} \mid \mathsf{C}) \approx_{\mathrm{f}} (\nu\,\mathsf{c})(\mathsf{F} \mid \mathsf{C}) \tag{10}$$

$$(\nu\,\mathsf{c})(\mathsf{F} \mid \mathsf{C}) \approx_{\mathrm{f}} (\nu\,\mathsf{c})(\mathsf{G} \mid \mathsf{D}) \tag{11}$$

The correctness of the whole transformations, *i.e.,* $\mathrm{SC} \approx_{\mathrm{f}} \mathrm{SC}' \approx_{\mathrm{f}} \mathrm{SC}'' \approx_{\mathrm{f}} \mathrm{SC}'''$ follows from closure under contexts from the equations above.

Note that the transformation of auxiliary communications (passing value w from r^{B} to s^{B} and value v from s^{B} to r^{B}) into normal communications are actually correct only up to weak full bisimilarity. In fact, auxiliary communications require a few more steps, and leave behind them empty sessions and streams, which have to be garbage collected. The correctness of garbage collection is based on Equations 1 and 2.

Proof (of Equation 9). The proof can be easily obtained by exhibiting a bisimulation including the two processes. For lack of space we will not show it, but just highlight a few important points. The two processes can mimic each other even if the first one is non deterministic, since the nondeterminism comes from τ steps, whose order is not important, since the processes are confluent. Also, as mentioned before, garbage collection Equations 1 and 2 are used in the proof. After some steps, the two processes have evolved to:

$$(\nu\,\mathsf{s})(r \rhd \mathsf{Q}[w/x][v'/y] \mid s \lhd \mathsf{R}[w/x][w'/z][v'/y] \mid s \rhd \mathsf{S}[w/x][v/y])$$

$$(\nu\,\mathsf{s})(r \rhd (s \lhd \mathsf{R}[w/x][w'/z][v'/y]) \mid \mathsf{Q}[w/x][v'/y] \mid s \rhd \mathsf{S}[w/x][v/y])$$

respectively. These processes can be proved equivalent using structural congruence (which is included in full bisimilarity, according to Lemma 2), session independence (Equation 3) and closure under contexts.

Proof (of Equation 10). To prove the correctness of Equation 10 it is enough to prove $E \approx_{\mathrm{f}} F$, then the thesis follows from closure under contexts. Actually, in general we can prove

$$(\nu\,\mathsf{a})(\mathcal{C}'[\![a \Uparrow v.P]\!] \mid \mathcal{C}''[\![a \Downarrow (y).Q]\!]) \approx_{\mathrm{f}} \mathsf{stream}\,\mathcal{C}'[\![\mathsf{feed}\,v.P]\!]\,\mathsf{as}\,f\,\mathsf{in}\,\mathcal{C}''[\![f(y).Q]\!] \tag{12}$$

provided that neither a nor f occur elsewhere and P and $\mathcal{C}'$ contain no feeds.

The proof shows that the three pairs below, together with a few other pairs differing from these because of τ transitions (corresponding to intermediate steps) form a full bisimulation.

$$\begin{array}{l}((\nu\,\mathsf{a})(\mathcal{C}'[\![a \Uparrow v.P]\!] \mid \mathcal{C}''[\![a \Downarrow (y).Q]\!]), \mathsf{stream}\,\mathcal{C}'[\![\mathsf{feed}\,v.P]\!]\,\mathsf{as}\,f\,\mathsf{in}\,\mathcal{C}''[\![f(y).Q]\!])\\ \quad (\mathcal{C}'[\![P]\!] \mid \mathcal{C}''[\![\mathsf{stream}\,\mathbf{0}\,\mathsf{as}\,f' = \langle v\rangle\,\mathsf{in}\,f'(y).Q]\!],\\ \qquad\qquad \mathsf{stream}\,\mathcal{C}'[\![P]\!]\,\mathsf{as}\,f = \langle v\rangle\,\mathsf{in}\,\mathcal{C}''[\![f(y).Q]\!])\\ \qquad (\mathcal{C}'[\![P]\!] \mid \mathcal{C}''[\![Q]\!], \mathsf{stream}\,\mathcal{C}'[\![P]\!]\,\mathsf{as}\,f\,\mathsf{in}\,\mathcal{C}''[\![Q]\!])\end{array}$$

In each process considered in the relation, a, f, and f' do not occur elsewhere, and P and $\mathcal{C}'$ do not contain feeds.

The only difficult part is when the feed and read from stream are executed (and, correspondingly, the two auxiliary communications). Actually both transitions amount to τ actions.

$$\mathsf{stream}\,\mathcal{C}'[\![\mathsf{feed}\,v.P]\!]\,\mathsf{as}\,f\,\mathsf{in}\,\mathcal{C}''[\![f(y).Q]\!] \xrightarrow{\tau} \mathsf{stream}\,\mathcal{C}'[\![P]\!]\,\mathsf{as}\,f = \langle v\rangle\,\mathsf{in}\,\mathcal{C}''[\![f(y).Q]\!]$$

$$\mathsf{stream}\,\mathcal{C}'[\![P]\!]\,\mathsf{as}\,f = \langle v\rangle\,\mathsf{in}\,\mathcal{C}''[\![f(y).Q]\!] \xrightarrow{\tau} \mathsf{stream}\,\mathcal{C}'[\![P]\!]\,\mathsf{as}\,f\,\mathsf{in}\,\mathcal{C}''[\![Q[v/y]]\!]$$

The first transition is as follows (where we used the definitions of auxiliary communications in the first step).

$$\begin{array}{l}(\nu\,\mathsf{a})\mathcal{C}'[\![a \Uparrow v.P]\!] \mid \mathcal{C}''[\![a \Downarrow (y).Q]\!] =\\ \qquad (\nu\,\mathsf{a})\mathcal{C}'[\![\mathsf{stream}\,a \Leftarrow v.\mathsf{feed}\,u\,\mathsf{as}\,f''\,\mathsf{in}\,f(x).P]\!] \mid\\ \qquad\qquad \mathcal{C}''[\![\mathsf{stream}\,a \Rightarrow (z)\mathsf{feed}\,z\,\mathsf{as}\,f'\,\mathsf{in}\,f(y).Q]\!] \xrightarrow{\tau}{}^{*}\\ (\nu\,\mathsf{a},\mathsf{r})\mathcal{C}'[\![\mathsf{stream}\,r \lhd \mathbf{0}\,\mathsf{as}\,f''\,\mathsf{in}\,P]\!] \mid \mathcal{C}''[\![\mathsf{stream}\,r \rhd \mathbf{0}\,\mathsf{as}\,f' = \langle v\rangle\,\mathsf{in}\,f(y).Q]\!] \sim_{\mathrm{f}}\\ \qquad\qquad \mathcal{C}'[\![P]\!] \mid \mathcal{C}''[\![\mathsf{stream}\,\mathbf{0}\,\mathsf{as}\,f' = \langle v\rangle\,\mathsf{in}\,f(y).Q]\!]\end{array}$$

where in the last step we used Equations 1 and 2. The second transition is matched by:

$$\begin{array}{l}\mathcal{C}'[\![P]\!] \mid \mathcal{C}''[\![\mathsf{stream}\,\mathbf{0}\,\mathsf{as}\,f' = \langle v\rangle\,\mathsf{in}\,f(y).Q]\!] \xrightarrow{\tau}\\ \qquad\qquad \mathcal{C}'[\![P]\!] \mid \mathcal{C}''[\![\mathsf{stream}\,\mathbf{0}\,\mathsf{as}\,f'\,\mathsf{in}\,Q[v/y]]\!] \sim_{\mathrm{f}} \mathcal{C}'[\![P]\!] \mid \mathcal{C}''[\![Q[v/y]]\!]\end{array}$$

where we used Equation 2 again. This concludes the proof.

Proof (of Equation 11). It is easy to verify that $(\nu\,\mathsf{c})(\mathsf{F} \mid \mathsf{C})$ can be typed according to the type system for sequentiality. By considering:

$$\mathcal{D}[\![\bullet_1, \bullet_2]\!] = b \Rightarrow (x)(\mathsf{stream}\ \bullet_1\ \mathsf{as}\,f\,\mathsf{in}\,f(y).y.Q) | \bullet_2$$

$$\mathcal{C}[\![\bullet]\!] = x.\bullet \qquad P = v.(y)\mathsf{feed}\, y.R$$
$$\mathcal{C}'[\![\bullet]\!] = (x)\bullet \qquad Q = (y)v'.S$$

we can apply Theorem 3 to get the thesis since prefixes (z) and w' have both index 2.

7 Conclusions

We have shown how to exploit formal techniques to define correct program transformations relating different styles of programming used in the field of service-oriented systems, namely object-oriented, session-based and request/request-response based. This allows to exploit the different techniques available in each field, and still get a system implemented using the desired technology.

In addition to that, we have illustrated the expressiveness of SSCC [10, 12], also in a field like object-oriented programming, for which it was not conceived. We have also demonstrated the benefit of working with sequential sessions, where more powerful transformations are available.

For future work we intend to investigate the possibility of extending the session breaking transformation to larger classes of systems. We are aware of the fact that parallel communications make the agreement between the client and the server on where to change session more difficult. Towards this end, a promising approach is to perform a preliminary transformation turning arbitrary sessions into sequential ones.

References

[1] Ambler, S.W.: The Object Primer: Agile Model-Driven Development with UML 2.0. Cambridge University Press, Cambridge (2004)
[2] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business Process Execution Language for Web Services, Version 1.1 (2003)
[3] Boreale, M., Bruni, R., Caires, L., De Nicola, R., Lanese, I., Loreti, M., Martins, F., Montanari, U., Ravara, A., Sangiorgi, D., Vasconcelos, V.T., Zavattaro, G.: SCC: a service centered calculus. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 38–57. Springer, Heidelberg (2006)
[4] Boreale, M., Bruni, R., De Nicola, R., Loreti, M.: Sessions and pipelines for structured service programming. In: FMOODS 2008. LNCS, Springer, Heidelberg (to appear, 2008)
[5] Bruni, R., Lanese, I., Melgratti, H., Tuosto, E.: Multiparty sessions in SOC. In: COORDINATION 2008. LNCS, Springer, Heidelberg (to appear, 2008)
[6] Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: SOCK: a calculus for service oriented computing. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 327–338. Springer, Heidelberg (2006)
[7] Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration conformance for system design. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 63–81. Springer, Heidelberg (2006)
[8] Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, Springer, Heidelberg (2007)
[9] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: WSDL: Web Services Definition Language. World Wide Web Consortium (2004)

[10] Cruz-Filipe, L., Lanese, I., Martins, F., Vasconcelos, V.T., Ravara, A.: Bisimulations in SSCC. DI/FCUL TR 07–37, Department of Informatics, Faculty of Sciences, University of Lisbon (2007)
[11] Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 22–138. Springer, Heidelberg (1998)
[12] Lanese, I., Martins, F., Vasconcelos, V.T., Ravara, A.: Disciplining orchestration and conversation in service-oriented computing. In: SEFM 2007, pp. 305–314. IEEE Computer Society Press, Los Alamitos (2007)
[13] Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 33–47. Springer, Heidelberg (2007)
[14] Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, Reading (1999)
[15] Sands, D.: Total correctness by local improvement in the transformation of functional programs. ACM Trans. Program. Lang. Syst. 18(2), 175–234 (1996)
[16] Sangiorgi, D.: Typed π-calculus at work: a correctness proof of Jones's parallelisation transformation on concurrent objects. Theory and Practice of Object Systems 5(1), 25–34 (1999)
[17] Sangiorgi, D., Walker, D.: The π-calculus: A theory of mobile processes. Cambridge University Press, Cambridge (2001)
[18] Sensoria project web site, `http://www.sensoria-ist.eu/`
[19] 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)
[20] Vieira, H.T., Caires, L., Seco, J.C.: The conversation calculus: A model of service oriented computation. In: ESOP 2008. LNCS, Springer, Heidelberg (to appear, 2008)
[21] Wirsing, M., Clark, A., Gilmore, S., Hölzl, M.M., Knapp, A., Koch, N., Schroeder, A.: Semantic-based development of service-oriented systems. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 24–45. Springer, Heidelberg (2006)

Mechanizing a Correctness Proof for a Lock-Free Concurrent Stack

John Derrick[1], Gerhard Schellhorn[2], and Heike Wehrheim[3]

[1]Department of Computing, University of Sheffield, Sheffield, UK
J.Derrick@dcs.shef.ac.uk
[2]Universität Augsburg, Institut für Informatik, 86135 Augsburg, Germany
schellhorn@informatik.uni-augsburg.de
[3]Universität Paderborn, Institut für Informatik, 33098 Paderborn, Germany
wehrheim@uni-paderborn.de

Abstract. Distributed algorithms are inherently complex to verify. In this paper we show how to verify that a concurrent lock-free implementation of a stack is correct by mechanizing the proof that it is linearizable, linearizability being a correctness notion for concurrent objects. Our approach consists of two parts: the first part is independent of the example and derives proof obligations local for one process which imply linearizabilty. The conditions establish a (special sort of non-atomic) *refinement relationship* between the specification and the concurrent implementation. These are used in the second part to verify the lock-free stack implementation. We use the specification language Z to describe the algorithms and the KIV theorem prover to mechanize the proof.

Keywords: Z, refinement, concurrent access, linearizability, non-atomic refinement, theorem proving, KIV.

1 Introduction

Locks have been used to control access by concurrent processors to shared objects and data structures. However, performance and other issues have led to the development of *lock-free* algorithms which allow multi-processors access to the data structures in a highly interleaved fashion. Such concurrent algorithms providing access to shared objects (e.g., stacks, queues, etc.) are intrinsically difficult to prove correct, and the down-side of the performance gain from using concurrency is the much harder verification problem: how can one verify that a lock-free algorithm is correct? This paper is concerned with the development of a theory that allows mechanized proof of correctness for lock-free algorithms.

As an example we use the lock-free stack from [19] which implements atomic push and pop operations as instructions to read, write and update local variables as well as the stack contents, the individual instructions being devised so that concurrent access can be granted. The only atomic operations are the reading and writing of variables and an atomic *compare-and-swap* (atomically comparing the values of two variables plus setting a variable). Sequential notions of correctness,

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 78–95, 2008.

such as refinement [5,22], which rely on strict atomicity being preserved for all operations in an implementation have to be adapted to provide a correctness criteria for such a concurrent non-atomic setting. Linearizability [13] is one such criteria which essentially says that any implementation could be viewed as a sequence of higher level operations. Like serializability for database transactions, it permits one to view concurrent operations on objects as though they occur in some sequential order [13]:

> Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response.

Recent work on formalizing and verifying lock-free algorithms includes [9,3,1,15] as well as our own [6]. These show correctness by showing that an abstraction (or simulation or refinement) relation exists between the abstract specification and the concurrent implementation [9,3,1,15,10,6]. The proofs are manual or partly supported by theorem provers like, for instance, PVS.

All these papers argue informally that refinement implies the original linearizability criterion of [13]. Our work instead gives a formal theory that relates refinement theory and linearizability, which has been fully mechanized using the interactive theorem prover KIV [20]. A web presentation of the KIV proofs is available [16].

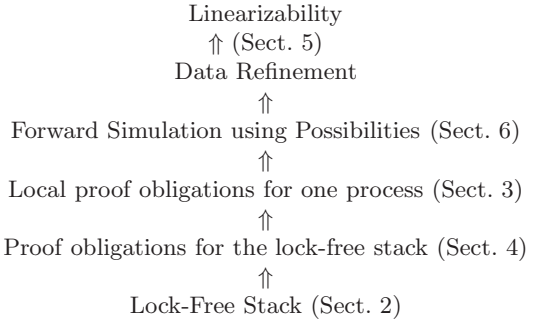


Fig. 1. Structure of the linearizability proof for the lock-free stack

Our methodology of proving linearizability consists of two parts: A generic part, shown in the upper half of Fig. 1 which derives proof obligations for one process. These proof obligations are shown to imply linearizability, and their genericity means they should be applicable to other algorithms in addition to the case study presented here. The second part is thus an application specific part (the lower half of Fig. 1) which instantiates these proof obligations for our particular case study.

We start with the lower half, which extends our work in [6], where we used non-atomic refinement to verify linearizability. The specifications given there were expressed as a combination of CSP and Object-Z. We only considered two

processes, one doing a *push* operation on the stack, and the other one a *pop* operation. In addition, we restricted the linearization point to always occur at the end of a sequence of operations implementing an atomic abstract operation. This allowed us to prove the correctness of a slightly simplified version of the algorithm given in [3].

Returning to this paper, in the next section we specify the stack and its lock-free implementation, given here as a Z specification rather than an integration of CSP and Object-Z (since this allows for simpler proof conditions). (The use of Z also allows the specification of several processes to be very elegantly captured in the specification using Z's notion of *promotion*, although we suppress this aspect here due to space restrictions.)

Section 3 then describes the background and the methodology that we have derived. It results in proof obligations that generalize the ones in [6] in two ways. First, we relax the assumption of having just two processes, and verify linearizability for an arbitrary numbers of processes. Second, we allow arbitrary linearization points. The proof obligations are applied on our running example in Section 4.

The second half of our paper is concerned with the top half of Fig. 1. In Section 5 we give a formal definition of linearizability and show that it can be viewed as a specific form of data refinement. Section 6 justifies our local proof obligations by constructing a global forward simulation using the possibilities employed in [13]. The last section concludes and discusses related work.

2 The Stack and Its Lock-Free Implementation

Our example stack and its lock-free implementation is based on that given in [3]. Initially the stack is described as a sequence of elements of some given type T together with two operations *push* and *pop*. *push* pushes its input $v?$ on the stack, and *pop* removes one element that is returned in $v!$. When the stack is empty, *pop* leaves it and returns the special value *empty* (assumed to be not in T). The specification is given in Z [21,2], with which we assume the reader is familiar. The abstract stack specification is defined by:

$$\begin{array}{|l} AS \\ \hline stack : \mathrm{seq}\ T \\ \hline \end{array}$$

$$\begin{array}{|l} AInit \\ \hline AS' \\ \hline stack' = \langle\,\rangle \\ \hline \end{array}$$

$$\begin{array}{|l} push \\ \hline \Delta AS \\ v? : T \\ \hline stack' = \langle v? \rangle \frown stack \\ \hline \end{array}$$

$$\begin{array}{|l} pop \\ \hline \Delta AS;\quad v! : T \cup \{empty\} \\ \hline stack = \langle\,\rangle \Rightarrow \\ \quad v! = empty \wedge stack' = stack \\ stack \neq \langle\,\rangle \Rightarrow \\ \quad v! = head\ stack \wedge stack' = tail\ stack \\ \hline \end{array}$$

The lock-free implementation uses a linked list of nodes which can be accessed by a number of processes. We use a free type to model linked lists so that a node is either *null* or consists of an element plus a further node (its successor in the list), we also define functions for extracting values from the nodes in a list:

$$Node ::= node\langle\langle T \times Node\rangle\rangle \mid null$$

$$first : Node \nrightarrow T \qquad \forall\, t : T, n : Node \bullet first\ node(t, n) = t$$

$$second : Node \nrightarrow Node \qquad \forall\, t : T, n : Node \bullet second\ node(t, n) = n$$

We will furthermore use a function $collect : Node \rightarrow \mathrm{seq}\, T$ later on, which collects all values of nodes reachable from some initial node in a list.

Informally a node consists of a value $val : T$ and a pointer *next*, which either points to the next node in the list or is empty and then has value *null*. A variable *head* is used to keep track of the current head of the list. Operations *push* and *pop* are split into several smaller operations, making new nodes, swapping pointers etc.. There is one operation atomically carrying out a comparison of values and an assignment: $CAS(mem, exp, new)$ (compare-and-swap) compares *mem* to *exp*; if this succeeds (i.e. *mem* equals *exp*), *mem* is set to *new* and *CAS* returns *true*, otherwise the *CAS* fails, leaves *mem* unchanged and returns *false*. In pseudo-code the push and pop operations that one process executes are given as follows (here, `head` refers to the head of the linked list):

```
push(v : T):
1  n:= new(Node);
2  n.val := v;
3  repeat
4     ss:= head;
5     n.next := ss;
6  until CAS(head,ss,n)
```

```
pop(): T:
1   repeat
2      ss:= head;
3      if ss = null then
4         return empty;
5      ssn := ss.next;
6      lv := ss.val
7   until CAS(head,ss,ssn);
8   return lv
```

Thus the *push* operation first creates a new node with the value to be pushed onto the stack. It then repeatedly sets a local variable *ss* to *head* and the pointer of the new node to *ss*. This ends once the final *CAS* detects that *head* (still) equals *ss* upon which *head* is set to the new node *n*. Note that the *CAS* in *push* does not necessarily succeed: in case of a concurrent pop, *head* might have been changed in between. The *pop* is similar: it memorizes the head it started with in *ss*, then determines the remaining list and the output value. If *head* is still equal to *ss* in the end, the *pop* takes effect and the output value is returned.

Previously we used CSP to describe the orderings of the individual operations as given in the pseudo-code above. However, it is sufficiently simple that we use a program counter of the form either a number (1) or an *O* or *U* (for pop or

push) followed by a number and give the ordering as part of the Z operations. Thus we get the following values for program counters:

$$PC ::= 1 \mid O2 \mid O3 \mid O5 \mid O6 \mid O7 \mid O8 \mid U4 \mid U5 \mid U6$$

We use a global state GS, that consists just of the head of the stack. This state is shared by all processes executing the algorithms of *push* and *pop* above:

GS

$head : Node$

GSInit

GS'

$head' = null$

Every process then possesses its own local state space (where local variables like `ss` and `n` reside) and the global shared data structure, i.e., the linked list characterised by its current head. The local state LS of a particular process consists of the variables given in the pseudo-code above together with a program counter *pc*:

LS

$sso, ssu, ssn, n : Node$
$pc : PC$
$lv : T \cup \{empty\}$

LSInit

LS'

$pc' = 1$

To distinguish their use in *pop* and *push* we have appended a u, o, respectively, to variables ss. Each line in the pseudo-code is turned into a Z operation, where the numbering is according to line numbers in the pseudo-code.

First the operations are described in terms of their effect for one process, i.e. in terms of the state GS and one state LS. We classify the operations into invoking operations (`INVOP`, the first operation in a *push* or *pop*) and return operations (`RETOP`). For instance *psh*2 is the invoking operation of *push*, constructing a new node.

*psh*2 [INVOP]

ΞGS
ΔLS
$v? : T$

$pc = 1 \wedge pc' = U4$
$n' = node(v?, null)$

*psh*4

ΞGS
ΔLS

$pc = U4 \wedge pc' = U5$
$ssu' = head$

*psh*5

ΞGS
ΔLS

$pc = U5 \wedge pc' = U6$
$n' = node(first\ n, ssu)$

$CAS_t psh$ [RETOP]

ΔGS
ΔLS

$pc = U6 \wedge pc' = 1$
$head = ssu$
$head' = n$

$CAS_f psh$

ΞGS
ΔLS

$pc = U6 \wedge pc' = U4$
$head \neq ssu$

As usual, the notation Ξ specifies that the respective state space is unchanged, whereas Δ allows for modifications. However, we adopt the Object-Z (as opposed to Z) convention that all variables not mentioned in the predicate of a schema stay the same. This simply makes the specifications much more readable (otherwise we would have to add a lot of predicates of the form $ssu' = ssu$ etc.) and has no semantic consequence. Note that the only operation within *push* modifying the global data structure is $CAS_t psh$ (the succeeding CAS), which is also the return operation.

Similarly, we model *pop*. The operation *pop*2 which reads *head* at line 2 is duplicated, since it is called as an invoking operation (when $pc = 1$) as well as when the loop is iterated ($pc = O2$).

$pop2inv$ [INVOP]
ΞGS
ΔLS
$pc = 1 \land pc' = O3$
$sso' = head$

$pop2$
ΞGS
ΔLS
$pc = O2 \land pc' = O3$
$sso' = head$

$pop3t$ [RETOP]
ΞGS
ΔLS
$v! : T \cup \{empty\}$
$pc = O3 \land pc' = 1$
$sso = null \land v! = empty$

$pop3f$
ΞGS
ΔLS
$pc = O3 \land pc' = O5$
$sso \neq null$

$pop5$
ΞGS
ΔLS
$pc = O5 \land pc' = O6$
$ssn' = second\ sso$

$pop6$
ΞGS
ΔLS
$pc = O6 \land pc' = O7$
$lv' = first\ sso$

$CAS_t pop$
ΔGS
ΔLS
$pc = O7 \land pc' = O8$
$head = sso$
$head' = ssn$

$CAS_f pop$
ΞGS
ΔLS
$pc = O7 \land pc' = O2$
$head \neq sso$

$pop8$ [RETOP]
ΞGS
ΔLS
$v! : T \cup \{empty\}$
$pc = O8 \land pc' = 1$
$v! = lv$

Here, we find two return operations: a *pop* can return with output *empty* (operation *pop*3*t*) or with the effect of one node actually removed from the front of the linked list and its value returned as an output (operation *pop*8).

All these operations are defined for the local state. For the full scenario, it is assumed that each operation is executed by processes $p \in P$, then working on GS and a local state $LS = lsf(p)$ returned by a function *lsf* that stores all local states. A formal definition of this full scenario could be given in terms of promotion (see [2] or [5]), for reasons of space we only give a simple relational definition which also adds histories in Section 5 (histories are needed for linearizabilty).

We now have an abstract model of the stack, where *push* and *pop* occurs atomically, and an implementation with several processes operating concurrently on

the linked list implementing the stack. The objective then is to show that this lock-free implementation is linearizable with respect to the initial abstract model. Technically, this is done showing a particular form of refinement relation between abstract and concrete specification.

3 The Refinement Methodology

To show that linearizability holds we take the local view of one process and define forward simulation conditions that show that the concrete implementation is a non-atomic refinement [7,8] of the abstract stack. The purpose of this type of non-atomic refinement is to show that the concrete system (with many small steps) resembles the abstract system with a smaller number of "larger" steps. We will argue informally, that the proof obligations are sufficient to guarantee linearizability, a formal justification will be given in Sections 5 and 6.

In a standard (atomic) refinement, an abstract operation is implemented by one concrete operation, and to verify the correctness of such a transformation forward (and backward) simulations are used to describe how the abstract and concrete specifications proceed in a step by step fashion. The simulations are given in terms of an *abstraction relation* R relating states of the concrete and abstract specification, and then one proves that from related states, every step of a concrete operation can be mimicked by a corresponding abstract operation leading to related states again. In the case of non-atomic refinement one abstract operation is now implemented by a number of concrete operations. To adapt the simulation conditions one requires that concrete steps are either matched by an abstract step or by no operation at all. One scenario of this kind is depicted in Figure 2.

In the diagram, the upper level shows states and transitions of the abstract specification, the lower those of the concrete system. Dashed lines depict the abstraction relation R. (The labellings will be explained later). Starting from a pair of related states, a number of steps in the concrete system might abstractly have no effect on the state, thus the abstraction relation R stays at the same abstract state while changing the concrete. However, some concrete transitions (in this case the one in the middle) match a corresponding abstract operation, and the usual condition in simulations tells us that an abstract operation needs to be executed such that the after-states are again related. Matching with an

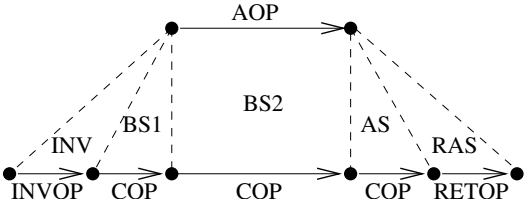


Fig. 2. Linearization in the middle

abstract step has to take place whenever the current concrete operation has a visible effect, and in our setting such visible effects are the linearization points. Details of how the non-atomic simulation conditions are derived and examples of their use are given in [7,8].

For our purposes we tailor the simulation conditions to our particular application of concurrent algorithms. First of all, since our concrete specification is partitioned into a global state GS together with local states LS for each process, we will describe the abstraction relation R as sets of elements $(as, gs, ls) \in R$ which we also write as $R(as, gs, ls)$. $R(as, gs, ls)$ then means that an abstract state as is related to a global concrete state gs and *one* local state ls [1].

The second adaption concerns the processes themselves. Processes can have three different states: they can be idle ($IDLE$), have already invoked the implementation of an abstract operation possibly with some input in ($IN(in)$) or they have already produced some output out for this operation ($OUT(out)$). This gives rise to the following types:

$$STATUS ::= IDLE \mid IN\langle\!\langle T \cup \{empty\}\rangle\!\rangle \mid OUT\langle\!\langle T \cup \{empty\}\rangle\!\rangle$$

To verify the simulation conditions, we require that a function $status : LS \to STATUS$ has been defined. We furthermore require the concrete set of operations that implement one abstract operation `AOP` to be split into three classes: *invocations* `INVOP`, *returns* `RETOP` and other operations (here denoted `COP`), as is the case in our stack example above. The formal KIV specification has concrete operations $\{COp_j\}_{j \in J}$ with J partitioned into IJ, RJ and CJ, abstract operations $\{AOp_i\}_{i \in I}$, and a mapping $abs : J \to I$ to define which concrete operation implements which abstract operation. For better readability we drop indices in the proof obligations.

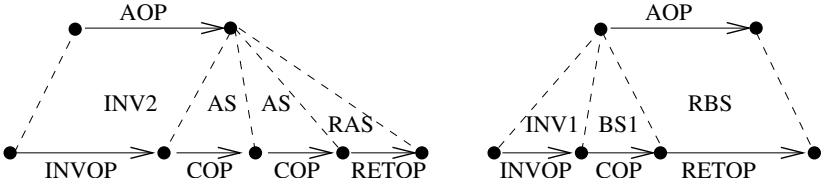


Fig. 3. Linearization on invocation or on return

The forward simulation conditions consist of six separate clauses: *Init, Invoke, Before Sync, After Sync, Return before sync* and *Return after sync*, which we describe in turn. In addition to the initialisation, these describe a number of different possibilities. First of all we have to say how invocations, returns and other operations behave. Second, we have to describe whether these operations match the abstract one (we call this a *linearization* with AOP in the descriptions below), or whether they should have no observable effect. This second aspect gives us the disjunctions in the conditions below.

[1] Note that we do not have all local states together in R.

First of all, **Init** is the standard initialisation condition, requiring that initial states of the two specifications are related:

Init

$$\exists\, as \in AInit \bullet R(as, gs, ls)$$

Like all following conditions, the formula is implicitly universally quantifies over all its free variables (here: $gs : GS$ and $ls : LS$). **Invoke** places a requirement on the operations marked as `INVOP`, which are the operations that begin a sequence of concrete operations corresponding to one abstract operation. In our example $psh2$ and $pop2inv$ are of type `INVOP`. Invoke requires that if a process is idle and an invoking operation is executed then one of two things happen. Either the concrete after states are linked to the same abstract states as before ($R(as, gs', ls')$, see diagram INV1 to the right of Figure 3) and the status is moved from $IDLE$ to IN, or that linearization has already taken place (diagram INV2 left of Figure 3), the process moves to its OUT state and we must match this invoke operation with the abstract AOP:

Invoke

$$\begin{array}{l} R(as, gs, ls) \wedge status(ls) = IDLE \wedge INVOP(gs, ls, gs', ls', in) \\ \quad \Rightarrow \quad (R(as, gs', ls') \wedge (\forall\, lsq : LS \bullet R(as, gs, lsq) \Rightarrow R(as, gs', lsq)) \\ \qquad\quad \wedge\, status(ls') = IN(in)) \\ \qquad \vee\, (status(ls') = OUT(out) \\ \qquad\quad \wedge \exists\, as' : AS \bullet AOP(in, as, as', out) \wedge R(as', gs', ls') \\ \qquad\qquad\qquad\quad \wedge\, (\forall\, lsq : LS \bullet R(as, gs, lsq) \Rightarrow R(as', gs', lsq)) \end{array}$$

An additional condition $\forall\, lsq : LS \bullet R(as, gs, lsq) \Rightarrow R(as', gs', lsq)$ is the price to pay for interleaving processes compared to standard non-atomic refinement: the condition prevents that other processes with an unknown, arbitrary state lsq are affected by the local operation.

Before sync describes what happens if a concrete COP is executed before we have reached the linearization point. It has two cases: either we have still not linearized after the COP, or this is the linearization point (see Figure 2), in which case we must match with the abstract operation AOP:

Before Sync

$$\begin{array}{l} R(as, gs, ls) \wedge status(ls) = IN(in) \wedge COP(gs, ls, gs', ls') \\ \quad \Rightarrow \quad (status(ls') = IN(in) \wedge R(as, gs', ls) \\ \qquad\quad \wedge\, (\forall\, lsq : LS \bullet R(as, gs, lsq) \Rightarrow R(as, gs', lsq))) \\ \qquad \vee\, (status(ls') = OUT(out) \\ \qquad\quad \wedge \exists\, as' : AS \bullet AOP(in, as, as', out) \\ \qquad\qquad\qquad\quad \wedge\, R(as', gs', ls') \\ \qquad\qquad\qquad\quad \wedge\, (\forall\, lsq : LS \bullet R(as, gs, lsq) \Rightarrow R(as', gs', lsq))) \end{array}$$

In our example, $psh4$, $psh5$, $CAS_f pop$, $pop3f$, $pop6$, $pop5$, $CAS_t pop$ are all of type **Before Sync**. A dual condition **After Sync** describes the effect of a concrete operation taking place after the linearization point:

After Sync

$$\begin{aligned} &R(as, gs, ls) \wedge status(ls) = OUT(out) \wedge COP(gs, ls, gs', ls') \\ &\quad \Rightarrow R(as, gs', ls') \wedge status(ls') = OUT(out) \\ &\qquad \wedge (\forall\, lsq : LS \bullet R(as, gs, lsq) \Rightarrow R(as, gs', lsq)) \end{aligned}$$

In our particular example, we do not have an operation that requires this condition.

The final two conditions describe the effect of the `RETOP` operations and there are two cases. We could either have linearized already (**Return after sync**, diagrams RAS in the figures) in which case the concrete operation should have no observable effect, but the output produced by `RETOP` has to match the current one in status (stored during a previous linearization), or we haven't already linearized, and **Return before sync** then says we must now linearize with the abstract operation AOP.

Return before sync

$$\begin{aligned} &R(as, gs, ls) \wedge status(ls) = IN(in) \wedge RETOP(gs, ls, gs', ls', out) \\ &\quad \Rightarrow \exists\, as' : AS \bullet AOP(in, as, as', out) \wedge R(as', gs', ls') \\ &\qquad\qquad \wedge status(ls') = IDLE \\ &\qquad\qquad \wedge (\forall\, lsq : LS : R(as, gs, lsq) \Rightarrow R(as', gs', lsq)) \end{aligned}$$

Return after sync

$$\begin{aligned} &R(as, gs, ls) \wedge status(ls) = OUT(out) \wedge RETOP(gs, ls, gs', ls', out') \\ &\quad \Rightarrow out' = out \wedge status(ls') = IDLE \wedge R(as, gs', ls') \\ &\qquad \wedge (\forall\, lsq : LS \bullet R(as, gs, lsq) \Rightarrow R(as, gs', lsq)) \end{aligned}$$

In our example, $CAS_t psh$ is of type **Return before sync**, whilst $pop8$ and $pop3t$ are of type **Return after sync**.

4 Application to the Stack Example

In this section we describe how the methodology of the previous section can be applied to our running example. This involves mainly working out the correct abstraction relation and the *status* function, and then verifying the conditions that we detailed above.

The representation relation is constructed out of several specific ones, detailing the particular local state at various points during the execution. In summary:

$$R = collect(head) = stack \wedge \bigvee\nolimits_{n \in PC} R^n \wedge pc = n$$

The overall R is the disjunction over individual R^ns, for n being some specific value of the program counter, conjoined with the predicate $pc = n$ and – the actual relationship between abstract and concrete state space – a predicate stating that collecting the values from `head` on gives us the stack itself. The individual R^ns capture information about the current local state of a process being at $pc = n$. For several ns no specific information is needed:

$$\begin{array}{|l} R^1 \\ \hline AS;\ GS;\ LS \\ \hline \end{array} \qquad R^{O2} \mathrel{\widehat{=}} R^1, R^{O3} \mathrel{\widehat{=}} R^1, R^{O8} \mathrel{\widehat{=}} R^1$$

For others, certain predicates on the local state are needed. For instance, R^{U4} states the existence of a locally created node, or R^{O6} the relationship between *sso* and *ssn*. There is no further specific relationship between abstract and concrete state known (except for $collect(head) = stack$, which is true everywhere). Even a previous read of *head* gives us no relationship to, e.g., *sso* since this may have been invalidated by other concurrently running processes.

$$\begin{array}{|l} R^{U4} \\ \hline LS \\ \hline \exists\, v, no' \bullet \\ \quad n = node(v, no') \\ \hline \end{array} \qquad \begin{array}{|l} R^{U5} \\ \hline LS \\ \hline \exists\, v, no' \bullet \\ \quad n = node(v, no') \\ \hline \end{array} \qquad \begin{array}{|l} R^{U6} \\ \hline LS \\ \hline \exists\, v \bullet n = node(v, ssu) \\ \hline \end{array}$$

$$\begin{array}{|l} R^{O5} \\ \hline LS \\ \hline sso \neq null \\ \hline \end{array} \qquad \begin{array}{|l} R^{O6} \\ \hline LS \\ \hline sso \neq null \\ ssn = second\ sso \\ \hline \end{array} \qquad \begin{array}{|l} R^{O7} \\ \hline LS \\ \hline sso \neq null \\ ssn = second\ sso \\ lv = first\ sso \\ \hline \end{array}$$

Furthermore we need the definition of *status*. The status of a local state essentially depends on the program counter, it determines whether a process is in state *IDLE*, *IN* or *OUT*, thereby determining the linearization points. Local variables n, *sso* and *lv* can be used to determine values for input and output.

$$\begin{array}{|l} status : LS \to STATUS \\ \hline \forall\, ls \in LS \bullet \\ \quad (ls.pc = 1 \Rightarrow status(ls) = IDLE) \\ \quad (ls.pc \in \{U4, U5, U6\} \Rightarrow status(ls) = IN(first\ ls.n)) \\ \quad (ls.pc = O3 \wedge ls.sso = null \Rightarrow status(ls) = OUT(empty)) \\ \quad (ls.pc \in \{O2, O5, O6, O7\} \vee (ls.pc = O3 \wedge ls.sso \neq null) \\ \qquad \Rightarrow status(ls) = IN(empty)) \\ \quad (ls.pc = O8 \Rightarrow status(ls) = OUT(ls.lv)) \end{array}$$

For our example, the linearization point of the push algorithm is the last returning instruction $CAS_t psh$ (so all intermediate values of the program counter

within push have status IN). For the pop algorithm it is also the last instruction $pop8$ for the case of a nonempty stack. But when $pop2$ finds an empty stack in $head$ and therefore sets sso' to $null$ this must already be the linearization point, since at any later point during the execution the stack might already be nonempty again. There $status$ is $OUT(empty)$ at $pc = O3$ and $sso = null$.

Finally, we look at the verification of one of our conditions. Here, we take a look at condition **Return after sync** and the case where the return operation is $pop8$. Thus, on the left side of our implication we have $R(as, gs, ls) \wedge status(ls) = OUT(out) \wedge pop8(gs, ls, gs', ls', out')$. Since $pop8$ is executed, we have $pc = O8$. Hence $status(ls) = OUT(lv)$. By definition of $pop8$, we furthermore get $gs = gs'$, $out' = lv$, $pc' = 1$. Hence $out' = out$ (first conjunct to be shown), $status(ls') = IDLE$ (follows from $pc' = 1$, second conjunct), and $R(as, gs', ls')$ holds by definition of R^1 ($pc' = 1$) and the previous validity of $collect(head) = stack$. Furthermore, from $R(as, gs, lsq)$ we get $R(as, gs', lsq)$ since $gs' = gs$. This can similarly be done for all conditions and all operations, but - as a manual verification is error prone - KIV was used for this purpose, which lead to several small corrections. For example, R^{U4} and R^{U5} originally contained (erroneously) the condition $\exists\, v \bullet n = node(v, null)$. Since the complexity of the linearizability proof is contained in the generic theory, and proof obligations are tailored to interleaved execution of processes, applying them is easy: specification and verification of the stack example needed two days of work.

5 Linearizability as Refinement

In this and the following section we show that our local proof obligations are sufficient to guarantee linearizability as defined in Herlihy and Wing's paper [13]. The process to do this is as follows: in this section we give two data types ADT and CDT with operations $\{AOp_{p,i}\}_{p\in P, i\in I}$ and $\{COp_{p,j}\}_{p\in P, j\in J}$. These are derived from the local operations $\{COp_j\}_{j\in J}$ and $\{AOp_i\}_{i\in I}$ used in the proof obligations by adding a second index $p \in P$ that indicates the process p executing the operation. For the concrete level, $COp_{p,j}$ now works on a local state $lsf(p)$ given by a function $lsf : P \to LS$ instead of LS. We also add histories to the states and operations on both levels, which are lists of invoke and return events. This is necessary, since the formal definition of linearizability is based on a comparison of two histories created by the abstract and concrete level. By placing this criterion in the finalization of ADT and CDT, we encode linearizability as a specific case of standard data refinement ([11], see [4] for the generalization to partial operations) between the two data types (the discrepancy between the sets of indices is bridged by function abs, as explained below).

Our proof obligations are then justified in the next section by showing that they imply a forward simulation FS between ADT and CDT. This is the most complex step in the verification, since the additional concept of possibilities is needed to define FS.

We start by defining the histories $H \in HISTORY$ that are maintained by the two data types. They are lists of invoke and return events $e \in EVENT$.

An event indicates that an invoke or return operation has taken place executed by a certain process with some input (invoke) or output (return). The formal definition is

$$EVENT ::= inv\langle\langle P \times I \times IN \rangle\rangle \mid ret\langle\langle P \times I \times OUT \rangle\rangle$$
$$HISTORY ::= seq(EVENT)$$

Predicates $inv?(e)$ and $ret?(e)$ check an event to be an invoke or a return. $e.p \in P$ is the process executing the event, $e.i$ denotes the index of the abstract operation to which the event belongs. For a history H, $\#H$ is the length of the sequence, and $H(n)$ its nth element (for $0 \leq n < \#H$). Executing operations adds events to a history. If an invoke operation $INVOP_j$ with input in is executed by process p, it adds $inv(p, i, in)$ to the history, where $i = abs(j)$ is the index of the corresponding abstract operation as given by function $abs : J \to I$ of Section 3. Using a function $lsf : P \to LS$ such that $lsf(p)$ is the local state of process p we therefore define $COp_{p,j}$ to be the operation

$$\begin{aligned} &COp_{p,j}(gs, lsf, H, gs', lsf', H') == \\ &\quad \exists\, ls' \bullet \quad INVOP_j(in, gs, lsf(p), gs, ls') \\ &\qquad\qquad \wedge\, H' = H \frown \langle inv(p, abs(j), in)\rangle \\ &\qquad\qquad \wedge\, lsf' = lsf \oplus \{p \mapsto ls'\} \end{aligned}$$

Similarly, $COp_{p,j}$ for a return operation $RETOP_j$ adds the corresponding return event to the history. Other operations leave the history unchanged.

The histories created by interleaved runs of processes form *legal* histories only: a legal history contains *matching pairs (mp)* of invoke and return events, for operations that have already finished, and *pending invocations (pi)*, where the operation has started (i.e. the invoke is already in the history), but not yet finished. Corresponding formal definitions are

$$\begin{aligned} mp(m, n, H) == \quad & m < n < \#H \wedge H(m).p = H(n).p \wedge H(m).i = H(n).i \\ & \wedge\, \forall\, k \bullet m < k < n \Rightarrow H(k).p \neq H(m).p \end{aligned}$$

$$pi(n, H) == inv?(H(n)) \wedge \forall\, m \bullet n < m < \#H \Rightarrow H(m).p \neq H(n).p$$

$$\begin{aligned} legal(H) == \forall\, n < \#H \bullet \textbf{if } & inv?(H(n)) \textbf{ then } pi(n, H) \vee \exists\, m \bullet mp(n, m, H) \\ & \textbf{else } \exists\, m \bullet mp(m, n, H) \end{aligned}$$

Histories created by abstract operations are *sequential*: each invoke is immediately followed by a matching return. A predicate $seq(HS)$ determines whether HS is a sequential history. Atomically executing AOp_i by process p adds both events to the sequential history. Therefore $AOp_{p,i}$ is defined as

$$\begin{aligned} &AOp_{p,i}(as, HS, as', HS') == \\ &\quad AOp_i(in, as, as', out) \wedge HS' = HS \frown \langle inv(p, i, in), ret(p, i, out)\rangle \end{aligned}$$

Linearizability compares a legal history H and a sequential history HS. For pending invokes the effect of the operation may have taken place (this will correspond

to $status = OUT$ in our proof obligations). For these operations corresponding returns must be added to H. Assuming these returns form a list H', all other pending invokes must be removed. Function *complete* removes pending invokes from $H \frown H'$. Formally we define

$$\begin{aligned} &linearizable(H, HS) == \\ &\quad \exists\, H' \subseteq ret? \bullet legal(H \frown H') \wedge seq(HS) \wedge lin(complete(H \frown H'), HS) \end{aligned}$$

where *lin* requires the existence of a bijection f between the $complete(H \frown H')$ and HS that is order-preserving: for two matching pairs $mp(m, n, H)$ and $mp(m', n', H)$, if the first operation finishes before the second starts (i.e., $n < m'$), the corresponding matching pairs in HS must be in the same order. This gives the following definitions

$$lin(H, HS) == \exists f \bullet inj(f, H, HS) \wedge surj(f, H, HS) \wedge presorder(f, H, HS)$$

$$\begin{aligned} inj(f, H, HS) == \forall\, m < n < \#H \bullet \quad & f(m) \neq f(n) \\ & \wedge (mp(m, n, H) \Rightarrow f(n) = f(m) + 1) \end{aligned}$$

$$surj(f, H, HS) == \forall\, m < \#H \bullet f(m) < \#HS \wedge HS(f(m)) = H(m)$$

$$\begin{aligned} presorder(f, H, HS) == \forall\, m, n, m', n' \bullet \\ mp(m, n, H) \wedge mp(m', n', H) \wedge n < m' \Rightarrow f(n) < f(m') \end{aligned}$$

Finally, we define a refinement between two data types $(CInit, \{COP_{p,j}\}, CFin)$ and $(AInit, \{AOP_{p,j}\}, AFin)$. Both initialization operations are required to set the history to the empty list. $AOp_{p,j}$ is $AOp_{p,abs(j)} \vee skip$ where $abs : J \to I$ maps the index of the concrete operation to the abstract operation it implements. The concrete finalization extracts the collected history and abstract finalization is the linearizability predicate:

$$\begin{aligned} &AFin(as, HS, H') == linearizable(H', HS) \\ &CFin(gs, lsf, H, H') == H' = H \end{aligned}$$

With these finalization operations linearizability becomes equivalent to data refinement for the two data types.

6 Possibilities and Forward Simulation

Reasoning with the definition of linearizability as given in the previous section is rather tricky, since it is based on occurrences of events (i.e. positions in the history list). This is the reason why all work we are aware of on linearizability does *not* use this definition, but other definitions, for example, those based on IO automata refinement such as [10] argue informally that these imply linearizability. The problem was already noticed by Herlihy and Wing themselves, and they gave an alternative definition based on *possibilities*. Possibilities require a set of operations AOp_i already, so they are less abstract than linearizability which only

uses events. The following rule set defines an inductive predicate $Poss(H, S, as)$[2] where H is a legal history, S is a set of returns that match pending invokes in H (those pending invokes for which the effect has already occurred), and as is an abstract state that can possibly be reached by executing the events in H (hence the term 'possibility').

$$\frac{ASInit(as)}{Poss(\langle\rangle, \varnothing, as)}\ Init \qquad \frac{Poss(H, S, as) \quad \forall\, m.pi(m, H) \Rightarrow H(m).p \neq p}{Poss(H \frown \langle inv(p, i, in)\rangle, S, as)}\ I$$

$$\frac{Poss(H, S, as) \quad \exists\, m.pi(m, H) \wedge H(m) = inv(p, i, in) \quad AOp_i(in, as, as', out)}{Poss(H, S \cup \{ret(p, i, out)\}, as')}\ S$$

$$\frac{Poss(H, S, as) \quad ret(p, i, out) \in S}{Poss(H \frown \langle ret(p, i, out)\rangle, S \setminus \{ret(p, i, out)\}, as)}\ R$$

Rule *Init* describes the possible initial states: No event has been executed, the abstract state is initial, and the set of returns is empty. Rule I allows to add an invoke event for process p, provided there is not already a pending one in H. Rule S corresponds to linearization points: the effect of the abstract operation takes place (which requires a pending invoke), and the return is added to the set S of returns. The return takes place in the last rule R, which adds the return to the history. Compared to the informal definition in [13], we had to add some explicit constraints which guarantee that all created histories are legal. Note that the rules for possibilities are a close match for the invoke–linearization point–return structure also present in our proof obligations.

One of the main tasks in formally justifying our proof obligations therefore is a proof that all possibilities are linearizable:

$$Poss(H, S, as) \Rightarrow \exists\, HS \bullet linearizable(H, HS)$$

Essentially this is Theorem 9 of [13]. For our own proof within KIV we needed to generalize the theorem to

$$Poss(H, S, as) \Rightarrow \exists\, HS \bullet \forall\, H' \bullet \\ eq(H', S) \Rightarrow legal(H \frown H') \wedge lin(complete(H \frown H'), HS)$$

where $eq(H', S)$ is true, iff H' is a duplicate free list that contains the same events as the set S. The proof is inductive over the number of applied rules. The case of the induction step, where rule S is applied is the most complex, since both $complete(H \frown H')$ and HS increase by adding a matching pair. Therefore the bijection between H and HS must change, which creates a large number of subcases to prove that the modified function is bijective and still preserves the

[2] The notation of the original definition is $(v, P, R) \in Poss(H)$. We use a predicate instead of a set, as instead of v for the abstract state and S instead of R to avoid confusion with the simulation relation. The parameter P of the original definition is redundant: P can be shown to be the set of pending invocations of H.

order of matching pairs. This proof, and the lemmas consumed, 10 days of the 15 days needed to do the KIV proofs for the whole case study.

Given that the existence of possibilities implies linearizability we are now able to justify our proof obligations. We provide a forward simulation FS, such that our proof obligations imply that $COp_{p,j}$ forward simulates $AOp_{p,abs(j)} \vee skip$ (where again, $abs : J \to I$ maps the index of the concrete operation to the one of the corresponding abstract operation). The definition of FS is based on relation R as used in our proof obligations and on possibilities

$$
\begin{array}{l}
FS(as, HS, gs, lsf, H) == \\
\quad (\forall\, p \bullet R(as, gs, lsf(p)) \\
\wedge \quad \{H(n) \bullet pi(n, H)\} \\
\quad = \{inv(p, i, in) \bullet runs(lsf(p)) = i \wedge status(lsf(p)) = IN(in)\} \\
\wedge \exists\, S \bullet \quad S = \{ret(p, i, out) \bullet runs(lsf(p)) = i \wedge status(lsf(p)) = out\} \\
\qquad\quad \wedge Poss(H, S, as) \\
\qquad\quad \wedge \forall\, H' \bullet (eq(H', S) \Rightarrow legal(H \frown H') \wedge lin(complete(H \frown H'), HS)
\end{array}
$$

This formula uses an auxiliary function *runs* defined on local states, that gives the index of the abstract operation, whose implementation is currently running[3]. It is a conjunction of three properties. The first requires that the relation R holds for all local states of processes. The second is an invariant for the concrete data type: the pending inputs in the history correspond exactly to those implementations which run operation i and have not yet passed the linearization point (i.e., status is $IN(in)$). The last conjunct gives the connection to possibilities. The set S consist of those return events where a process p is running operation i and has reached status $OUT(out)$. The last line of the formula should be compared to the proof that possibilities imply linearizability above. There, a possibility implied the existence of a suitable HS with this property. This HS is now the HS constructed by the corresponding abstract run.

The proof that the given formula is indeed a forward simulation is moderately complex. The proof of the main commutativity property for correctness splits into three cases for invoking, return and other operations. Condition **Invoke** is needed for the first case. Internal operations require **Before Sync** for the case where status is $IN(in)$, and **After sync** otherwise. Return operations similarly require **Return before sync** and **Return after sync** respectively.

7 Conclusion

In this paper we have considered the verification of correctness of a lock-free concurrent algorithm. In a state-based setting we have followed the approach of using simulations to show that a linearizability condition is met. The correctness proof involved several steps, showing first of all the existence of a particular simulation relation between concrete algorithm and abstract specification and furthermore, via a number of steps, proving that such a simulation relation

[3] In our example, *runs* indicates whether the *pc* of p is within a *push* or *pop* operation.

implies linearizability. All proof steps have been mechanically checked using the theorem prover KIV.

Similar work on showing linearizability has been done by Groves and several co-authors [9,3]. In [3] specifications are written as IO-automata and linearization is shown using forwards and backwards simulation, and these were mechanized in PVS. The mechanization however only included the concrete case study. That simulation guarantees linearizability is argued but not mechanized.

Further work by Groves and Colvin includes [10], where they verify an improved version of an algorithm of Hendler et al. [12] which in turn extends the algorithm of [3] using a new approach based on action systems. This approach, like ours, starts with an abstract level of atomic push and pop operations. The approach uses a different proof technique than ours and their earlier work. Specifically, it works by induction over the number of completed executions of abstract operations contained in a history, and it is based on the approaches of Lipton [18] and Lamport and Schneider [17].

Additional relevant work in state-based formalisms includes [1], where the correctness of a concurrent queue algorithm using Event_B is shown. There, instead of verifying a given implementation, correctness is achieved by construction involving only correct refinement steps.

Another strand of relevant work is that due to Hesselink who has considered the verification of complex non-atomic refinements in the setting of a refinement calculus based notation. For example, in [14] he specifies and verifies (using PVS) a refinement of the lazy caching algorithm, where the model is not linearizable but only sequentially consistent. In [15] he recasts linearizability proofs as a refinement proof between two models which are verified by *gliding simulations* which allow the concrete model to do fewer steps than the abstract one if necessary.

The emphasis of our work was on the derivation of a generic, non-atomic refinement theory for interleaved processes and its mechanization. We have not yet considered the full complexity of the case study as done in [10] which adds memory allocation, modification counts to avoid the ABA problem and extends the algorithm by adding elimination arrays. These extensions remain as further work.

Acknowledgements. John Derrick and Heike Wehrheim were supported by a DAAD/British Council exchange grant for this work. The authors like to thank Lindsay Groves for pointing out an erroneous choice of linearization point in an earlier work and for many helpful comments when preparing this paper.

References

1. Abrial, J.-R., Cansell, D.: Formal Construction of a Non-blocking Concurrent Queue Algorithm (a Case Study in Atomicity). Journal of Universal Computer Science 11(5), 744–770 (2005)
2. Barden, R., Stepney, S., Cooper, D.: Z in Practice. BCS Practitioner Series. Prentice-Hall, Englewood Cliffs (1994)

3. Colvin, R., Doherty, S., Groves, L.: Verifying concurrent data structures by simulation. ENTCS 137, 93–110 (2005)
4. de Roever, W., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge Tracts in Theoretical Computer Science, vol. 47. Cambridge University Press, Cambridge (1998)
5. Derrick, J., Boiten, E.: Refinement in Z and Object-Z: Foundations and Advanced Applications. Springer, Heidelberg (2001)
6. Derrick, J., Schellhorn, G., Wehrheim, H.: Proving linearizability via non-atomic refinement. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 195–214. Springer, Heidelberg (2007)
7. Derrick, J., Wehrheim, H.: Using coupled simulations in non-atomic refinement. In: Bert, D., P. Bowen, J., King, S. (eds.) ZB 2003. LNCS, vol. 2651, pp. 127–147. Springer, Heidelberg (2003)
8. Derrick, J., Wehrheim, H.: Non-atomic refinement in Z and CSP. In: Treharne, H., King, S., C. Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, Springer, Heidelberg (2005)
9. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004)
10. Groves, L., Colvin, R.: Derivation of a scalable lock-free stack algorithm. ENTCS (to appear, 2007)
11. Jifeng, H., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP 1986. LNCS, vol. 213, pp. 187–196. Springer, Heidelberg (1986)
12. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: SPAA 2004: ACM symposium on Parallelism in algorithms and architectures, pp. 206–215. ACM Press, New York (2004)
13. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
14. Hesselink, W.H.: Refinement verification of the lazy caching algorithm. Acta Inf. 43(3), 195–222 (2006)
15. Hesselink, W.H.: A criterion for atomicity revisited. Acta Inf. 44(2), 123–151 (2007)
16. Web presentation of the linearization case study in KIV. URL: `http://www.informatik.uni-augsburg.de/swt/projects/linearizability.html`
17. Lamport, L., Schneider, F.B.: Pretending atomicity. Technical Report TR89-1005, SRC Digital (1989)
18. Lipton, R.J.: Reduction: a method of proving properties of parallel programs. Commun. ACM 18(12), 717–721 (1975)
19. Michael, M.M., Scott, M.L.: Nonblocking algorithms and preemption-safe locking on multiprogrammed shared — memory multiprocessors. Journal of Parallel and Distributed Computing 51(1), 1–26 (1998)
20. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV (ch. 1: Interactive Theorem Proving). In: Bibel, W., Schmitt, P. (eds.) Automated Deduction—A Basis for Applications. Systems and Implementation Techniques, vol. II, pp. 13–39. Kluwer Academic Publishers, Dordrecht (1998)
21. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall, Englewood Cliffs (1992)
22. Woodcock, J.C.P., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Englewood Cliffs (1996)

Symbolic Step Encodings for Object Based Communicating State Machines*

Jori Dubrovin, Tommi Junttila, and Keijo Heljanko

Helsinki University of Technology (TKK)
Department of Information and Computer Science
P.O. Box 5400, FI-02015 TKK, Finland
{Jori.Dubrovin,Tommi.Junttila,Keijo.Heljanko}@tkk.fi

Abstract. In this work, novel symbolic step encodings of the transition relation for object based communicating state machines are presented. This class of systems is tailored to capture the essential data manipulation features of UML state machines when enriched with a Java-like object oriented action language. The main contribution of the work is the generalization of the $\exists$-step semantics approach, which Rintanen has used for improving the efficiency of SAT based AI planning, to a much more complex class of systems. Furthermore, the approach is extended to employ a dynamic notion of independence. To evaluate the encodings, UML state machine models are automatically translated into NuSMV models and then symbolically model checked with NuSMV. Especially in bounded model checking (BMC), the $\exists$-step semantics often significantly outperforms the traditional interleaving semantics without any substantial blowup in the BMC encoding as a SAT formula.

1 Introduction

This paper describes a method that allows UML state machine models enriched with a Java-like object oriented action language to be efficiently model checked with symbolic model checking techniques. We describe the theoretical background of a tool that encodes the transition relation of these object based communicating state machines in the NuSMV [1] model checker input language in a novel way that, in particular, makes the search for short counterexamples competitive with the state-of-the-art explicit state model checker Spin [2].

The model checking approach our encoding is best suited for is bounded model checking (BMC) [3] that has been introduced as an alternative to binary decisions diagrams (BDDs) to implement symbolic model checking. The main contributions of our work are symbolic *step semantics* encodings of the transition relation for object based communicating state machines. Similarly to partial order reduction techniques such as stubborn, ample, persistent, or sleep sets for

* Work financially supported by Helsinki Graduate School in Computer Science and Engineering, Tekes — Finnish Funding Agency for Technology and Innovation, Nokia, Conformiq Software, Mipro, the Academy of Finland (projects 112016, 211025, and 213113), and Technology Industries of Finland Centennial Foundation.

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 96–112, 2008.

explicit state model checking (see e.g., [4]) the idea is to exploit the concurrency available in the analyzed system to make the model checking for it more efficient. The main idea in the above mentioned partial order reduction methods is to try to generate a reduced state space of the system by removing some of the edges of the state space while still preserving the property (such as the existence of a deadlock state) being verified. However, the considerations for BMC are usually quite different from explicit state model checking. Instead of removing edges from the state space trying to minimize the size of the reduced state space, the idea here is to try to minimize the bound needed to reach each state of the system. Our approach will not remove any edges but will instead add a number of "shortcut edges" to the state space of the analyzed system, intuitively executing several actions at the same time step, as allowed by the concurrency of the system being analyzed. The hope is that by making more states reachable with smaller bounds, the worst case exponential behavior of the bounded model checker wrt. the bound k can be alleviated by allowing bugs to be found with smaller values of k. The decrease in the bound k needs of course to be balanced against the size of the transition relation encoding as well as the efficiency of the SAT checker in solving the generated BMC instances.

As the system model we consider object based communicating state machines, a model tailored to capture the essential data manipulation features of UML state machines when enriched with a Java-like object oriented action language. The work aims at analyzing object oriented data communications protocol software designed using UML state machines, see [5]. The model checking tool we have developed can also handle other aspects of UML state machines not covered in this paper for the sake of clarity, see Sect. 4 for details.

Our encoding is a significant generalization of the approach of Rintanen [6] on AI planning, where the notion of $\exists$-step semantics for planning problems was first systematically employed. There are the following substantial differences to this earlier work. First of all, our setup employs a UML state machine based model of concurrency enriched with asynchronous message passing and full object based data handling features such as dynamic object references. We show how the $\exists$-step semantics can still be efficiently encoded with a full Java-like action language. The main challenge is indeed the handling of the complex action language parts of the encoding, something that is non-existent in the AI planning domain. Secondly, our approach also introduces the novel use of a dynamic dependency relation to optimize the encoding even further. This allows for example concurrent attribute access of different instances of the same object at the same time step. The approach of Rintanen is based on a static dependency relation.

Other Related Work. In the area of SAT based BMC, Heljanko was the first to consider exploiting the concurrency in encoding the transition relation [7]. That paper considers BMC for 1-bounded Petri nets using the $\forall$-step semantics (the classical Petri net step semantics, see e.g., [8]). The intuitive idea is that a set of actions can be executed at the same time step in $\forall$-step semantics if they can be executed in all possible orders. In the area of SAT based AI planning already

the early papers of Kautz and Selman used ∀-step semantics [9] (see also [6]). The work of Dimopoulos et al. was the first one to use an ∃-step semantics like approach in hand optimized planning encodings of [10], the idea of which was later formalized and automated in the SAT based planning system of Rintanen et al. [6,11]. On the BMC side, Ogata et al. [12] and Jussila et al. [13,14] both show other approaches to obtaining an optimized transition relation encoding for 1-bounded Petri nets and labeled transition systems (LTSs), respectively. A nice overview of many optimized transition relation encodings for LTSs is the doctoral thesis of Jussila [15].

2 Systems and Semantics

In this paper we consider a class of systems which are composed of a finite set of asynchronously executing objects communicating with each other through message passing and data attribute access. The behavior of each object is defined by a state machine. For instance, Figs. 1(a)–(c) show a part of a simple heart beat monitor system described in UML state machines with Java-like action language annotations. The dynamic behavior of a system is captured by its *interleaving state space*

$$M = \langle C, c_{\text{init}}, \Delta \rangle,$$

where C is the set of all *global configurations*, $c_{\text{init}} \in C$ is an *initial global configuration*, and the *transition relation* $\Delta \subseteq C \times A \times C$ describes how configurations may evolve to others: $\langle c, a, c' \rangle \in \Delta$ iff the configuration c can change to c' by executing an action $a \in A$. As an example, Fig. 1(d) shows a part of the state space (ignore the dashed arrow for a while) of the system in Figs. 1(a)–(c); if the action $\langle o_1, t_{22} \rangle$ (corresponding to object o_1 firing its transition t_{22}) is executed in configuration c_1, it changes to c_2. We say that a configuration c'' is *reachable* from a configuration c if there exist $a_1, \ldots, a_k$ and $c_0, c_1, \ldots, c_k$ for some $k \in \mathbb{N}$ such that (i) $c = c_0$, (ii) $\forall 1 \leq i \leq k : \langle c_{i-1}, a_i, c_i \rangle \in \Delta$, and (iii) $c_k = c''$.

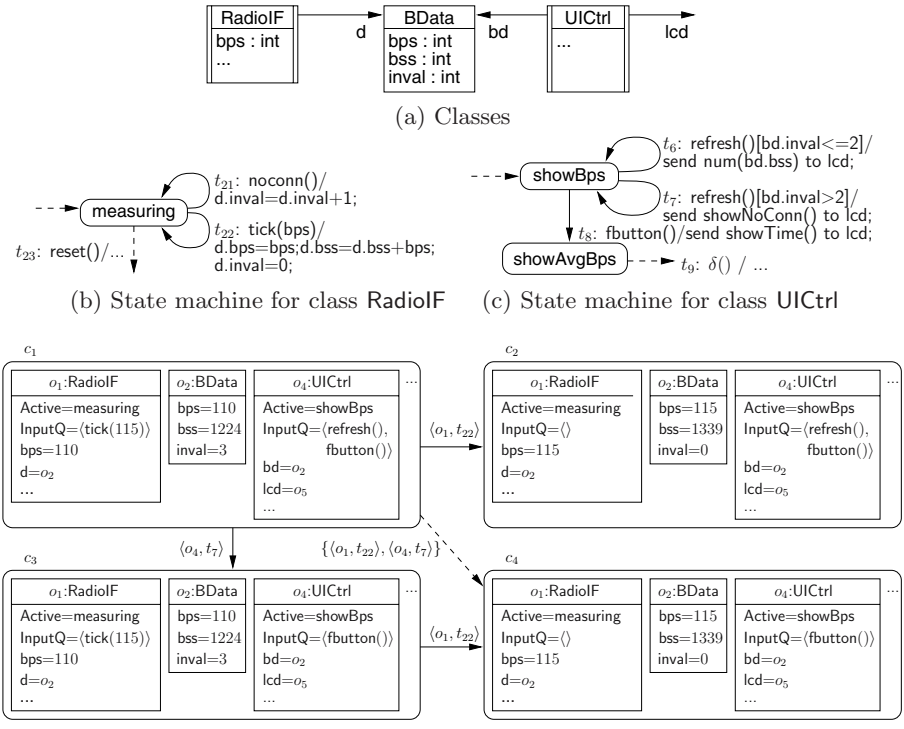
The basic idea in ∃-step semantics exploited in this paper is to augment the state space with "shortcut edges" so that, under certain conditions, several actions can be executed "at the same time step". This is formalized in the definition below.

Definition 1. *The* ∃-step state space *corresponding to the interleaving state space* $M = \langle C, c_{\text{init}}, \Delta \rangle$ *is the tuple*

$$M_\exists = \langle C, c_{\text{init}}, \Delta_\exists \rangle$$

where the transition relation $\Delta_\exists \subseteq C \times 2^A \times C$ *contains a* step $\langle c, S, c' \rangle$ *iff*

1. *the set of actions* $S = \{a_1, \ldots, a_k\}$ *is finite and non-empty, and*
2. *there is a total ordering* $a_1 \prec \cdots \prec a_k$ *of* S *and configurations* $c_0, c_1, \ldots, c_k \in C$ *such that (i)* $c = c_0$*, (ii)* $\forall 1 \leq i \leq k : \langle c_{i-1}, a_i, c_i \rangle \in \Delta$*, and (iii)* $c_k = c'$*.*



(a) Classes

(b) State machine for class RadioIF (c) State machine for class UICtrl

(d) A part of the state space with three actions and one shortcut edge

Fig. 1. A part of a simple heart beat monitor system

Continuing the running example, the dashed arrow in Fig. 1(d) denotes the step $\langle c_1, S, c_4\rangle$ with $S = \{\langle o_1, t_{22}\rangle, \langle o_4, t_7\rangle\}$. The actions in the step can be executed in the order $\langle o_4, t_7\rangle \prec \langle o_1, t_{22}\rangle$ but not in $\langle o_1, t_{22}\rangle \prec \langle o_4, t_7\rangle$ as executing $\langle o_1, t_{22}\rangle$ disables $\langle o_4, t_7\rangle$.

By definition it holds that the $\exists$-step state space includes the interleaving state space in the sense that $\langle c, a, c'\rangle \in \Delta$ implies that the *unit step* $\langle c, \{a\}, c'\rangle$ is in $\Delta_\exists$. Conversely, if $\langle c, S, c'\rangle$ belongs to $\Delta_\exists$, then there is a finite sequence of configurations leading from c to c' in the interleaving state space. Therefore, the set of configurations reachable from a given configuration is equal for the interleaving and the $\exists$-step state space. Note that the definition of $\exists$-step semantics is a purely semantic one; in the symbolic encoding given later, not all possible non-unit steps will be considered but only those that follow conveniently without complicating and growing the size of the encoding too much w.r.t. the one for the interleaving semantics. For example, similarly to [6], we require all actions of a step to be enabled already in the current configuration $c = c_0$. However, all unit steps will be included in order to preserve the soundness and completeness of the resulting encoding for the purpose of checking the reachability of desired/unwanted configurations. For complexity results on the semantic definition of $\exists$-step semantics in

the AI planning domain, see [6], which also similarly soundly underapproximates the ∃-step semantics in its implementation.

2.1 Object Based State Machine Models

We next give a brief description of the class of systems analyzed in this paper. Refer to the technical report version of this paper [16] for the formal definitions.

We consider systems composed of a fixed and finite set O of *objects*. Each object is an instance of a *class*, and each class is composed of a finite set of typed *attributes* and a *state machine*. A state machine consists of a finite set of *states*, one of which is the *initial state*, and a finite set of *transitions*. Each transition has a *source state* and a *target state*, a *trigger*, a *guard*, which is an action language expression of Boolean type, and an *effect*, which is a list of action language statements. A trigger is of the form $sig(x_1, \ldots, x_k)$, where *sig* is a *signal* from a finite set *Sigs*, and the x_i are attributes of the class owning the state machine. The number and types of the x_i must match the predefined parameter types of the signal. A special signal $\delta \in Sigs$ with no parameters models spontaneously triggered transitions. For example, state measuring is both the source and the target of transition t_{22} in Fig. 1(b). The trigger of t_{22} is tick(bps), and the guard of t_{22} is implicitly `true`.

Objects can send and receive *messages* of the form $sig[v_1, \ldots, v_k]$, where $sig \neq \delta$ is a signal and the values v_i are message *arguments*. The number and types of arguments correspond to the parameter types of *sig*. We denote the set of all messages by *Msgs*. During execution, messages sent to an object are placed in a queue, and an object can consume a message from its queue either by firing a transition triggered by the corresponding signal or by *implicit consumption*, which means discarding a message that is not triggering any transitions. Spontaneously triggered transitions are fired without consuming messages.

A global configuration c of the system consists of, for each object o, (i) the currently *active* state of o, which is one of the states in the state machine of the class of o, (ii) the value of the instance $o.x$ of each attribute x of the class of o, and (iii) the contents of the input queue of o, which is a sequence of messages. We will call the frontmost (oldest) message in the queue the *current* message of o. The initial configuration c_{init} is such that all objects are in their initial states and all input queues are empty.

The actions of the system are the *transition instances* $\langle o, t \rangle$ and the *implicit consumption actions* $\langle o, \text{IMPCONS} \rangle$, where o is an object and t is a transition in the state machine of o. A transition instance $\langle o, t \rangle$ is *enabled* in a global configuration c if (i) the source state of t is active in o, (ii) the guard of t evaluates to true in the context of o, and (iii) either the trigger of t is $\delta()$ or o has a current message whose signal matches the trigger signal of t. Of the global configurations in Fig. 1(d), the transition instance $\langle o_4, t_6 \rangle$ is only enabled in c_2, in which the current message of o_4 is refresh() and $o_2.\text{inval} \leq 2$.

An enabled action can be *executed* in a global configuration. Firing transition t in object o, or more formally, executing an enabled transition instance $\langle o, t \rangle$ in a

global configuration c, leads to a new global configuration c' that is obtained from c by (i) assigning the argument values of the current message of o to the attributes mentioned in the trigger of t, (ii) removing the current message from the input queue of o, (iii) executing the effect of t in the context of o, and (iv) making the target of t the new active state of o. If t is a spontaneously triggered transition, i.e. the trigger of t is $\delta()$, then points (i) and (ii) above are not performed.

An implicit consumption action $\langle o, \text{IMPCONS}\rangle$ is enabled if (i) the input queue of o is not empty, and (ii) there is no enabled transition instance $\langle o, t\rangle$ such that the trigger of t is not $\delta()$. Executing an enabled implicit consumption action $\langle o, \text{IMPCONS}\rangle$ in a global configuration c leads to a global configuration c' that is equal to c except that the current message of o is removed.

The interleaving state space is now defined as the tuple $M = \langle C, c_{\text{init}}, \Delta\rangle$, where C is the set of all global configurations, $c_{\text{init}} \in C$ is the initial configuration, and the transition relation $\Delta \subseteq C \times A \times C$ consists of the triples $\langle c, a, c'\rangle$ such that the transition instance or implicit consumption action a is enabled in c and executing a in c leads to c' by the above rules.

Action Language. In the sequel, we fix a simple Java-based action language [17] for expressing the guards and effects of transitions. The type system consists of the Boolean type $\mathbb{B}$, the 32-bit signed integer type, and for each class C the reference type $\mathbb{T}_C$ with domain $\{o \in O \mid \text{class of } o \text{ is } C\} \cup \{\textbf{null}\}$. As in Java, static typing rules apply. The supported expressions are: (i) literals `true`, `false`, `null`, and 32-bit signed integer literals, (ii) the `this` reference, (iii) attribute access expressions of the form *refexpr*`.`x, where the type of *refexpr* is $\mathbb{T}_C$ and x is an attribute of class C, and (iv) infix expressions *leftexpr* *op* *rightexpr*, where *op* is one of `+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`, `>`, `<`, `>=`, `<=`, `==`, or `!=`, with Java semantics [18]. The only data accessible to expressions is attribute values reachable by following references. In particular, an expression cannot read the active state or the input queue of any object. An unqualified attribute name x is shorthand for `this.`x.

The kinds of statements of the language are: (i) assignments of the form *refexpr*`.`x `=` *rhsexpr*`;`, (ii) send statements of the form `send` *sig*`(`$arg_1, \ldots, arg_k$`)` `to` *targetexpr*`;`. When a send statement is executed, the input queue of the object referred by *targetexpr* is appended with the message $sig[v_1, \ldots, v_k]$, where each v_i is the value of arg_i, and (iii) assertions of the form `assert` *condexpr*`;`. We want to check that *condexpr* is never false when an assertion is executed.

The effect of a transition is an arbitrary list of statements. However, we require that for each transition t and class C, there is at most one send statement in the effect of t whose *targetexpr* has type $\mathbb{T}_C$. The reason for this is that the symbolic encoding relies on the fact that in one step, at most one message is sent to each object.

Method calls are not directly supported, but a non-recursive call can emulated by either inlining it at the call site or by modeling it as a pair of asynchronous message transmissions, one for the invocation and one for the return.

3 Symbolic Encoding

The encoding of a transition relation is based on *constraints* involving *state variables*, whose valuation represents a global configuration, *next-state variables*, whose valuation represents the global configuration after a step, *input variables* whose values are only limited by the constraints, and *derived functions* that are defined over the variables. The basic idea is that all constraints are satisfied if and only if there is a step (in a subset of $\Delta_\exists$) from the configuration represented by the values of state variables to the configuration represented by the next-state variables. The desired properties of the system are encoded as a set of *invariants* that are to be verified using model checking. Many of the variables and functions have values in the Boolean domain. Non-Boolean variables and functions have a finite domain and thus can be booleanized to enable the use of SAT- and BDD-based techniques.

To keep the state space finite, we restrict the analysis to *bounded global configurations*, setting an upper limit QSIZE to the number of messages in any queue. Let $M = \langle C, c_{\text{init}}, \Delta \rangle$ be an interleaving state space, and let C^B be the set of configurations $c \in C$ such that the length of the input queue of o in c is at most QSIZE for all objects o. The *bounded interleaving state space* is $M^B = \langle C^B, c_{\text{init}}, \Delta^B \rangle$, where $\Delta^B = \{\langle c, a, c' \rangle \in \Delta \mid c, c' \in C^B\}$. As the semantics of Sect. 2.1 allows queues to grow indefinitely, by this restriction we effectively disregard all executions of the system where any queue at any point contains more than QSIZE messages. Consequently, some reachable configurations of the system may be omitted in the analysis. Whether or not such omission occurs could be detected by adding a suitable invariant to check, and if needed, QSIZE could be incremented to cover a larger set of the reachable configurations.

Fixed Ordering of Actions in Steps. We fix an arbitrary total order $\prec$ of the set of actions A. The intuitive idea is that instead of considering all possible orderings of actions as the $\exists$-step semantic definition allows, $\prec$ gives us one static order in which the executability of actions in a step will be guaranteed. To do this, the encoding must capture the state changes done by each action, and disallow all steps where a value modified by some action is (explicitly or implicitly) read by an action that is greater wrt. the order $\prec$. By doing this, we ensure that all of the $\exists$-steps allowed by the encoding are executable in the order given by $\prec$.

We will thus get a different encoding for each choice of $\prec$. The only requirement is that if $o_1, o_2 \in O$ and t_2 is a transition in the state machine of o_2, then $\langle o_1, \text{IMPCONS} \rangle \prec \langle o_2, t_2 \rangle$ must hold. In words, all implicit consumption actions must precede all transition instances in the total order. The reason for this will be discussed in Sect. 3.4.

The choice of a good total order is an interesting question that we have left for further study. A set of actions is *independent* if no action reads any part of the system state modified by another action. Notice that an independent set of enabled actions can always be executed in all orders, and thus the $\exists$-step

semantics always allows a set of independent actions to form an ∃-step for any selection of the total order $\prec$.

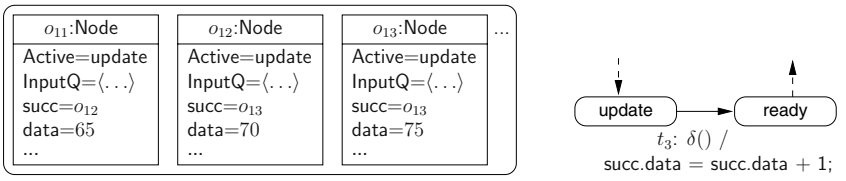
Step Encoding Overview. At a very coarse level, the encoding of ∃-steps contains three sets of constraints as follows.

1. For all actions a and all attributes $\hat{x}$: if a is executed and it writes to $\hat{x}$, then the value of $\hat{x}$ in the next global configuration is equal to the written value as if a was executed alone.
2. For all attributes $\hat{x}$: if no executed action writes to $\hat{x}$, then its value remains unchanged in the next global configuration.
3. For all actions a^- and a, and all attributes $\hat{x}$: if both a^- and a are executed and $a^- \prec a$ and a^- writes to $\hat{x}$, then a does not read from $\hat{x}$.

Constraint sets 1 and 2 together correspond to an encoding of *parallel* execution of actions. Any non-conflicting set of enabled actions can be executed at the same time, and the written values are computed independently for each action. Several actions are allowed to write to the same attribute at the same step, but only if the value written by each action is the same. For example, if action a_1 has the effect `o.x = o.y;` and action a_2 has the effect `o.x = o.y + 1;`, they cannot both be executed because there is no next-state value of `o.x` that satisfies the constraints. However, the parallel encoding allows the concurrent execution of a_1 together with another action a_3 whose effect is `o.y = o.x;`. Such a step swaps the values of `o.x` and `o.y`. This does not correspond to the sequential execution of a_1 and a_3 in either order, and thus we want to rule out non-∃-steps like this.

For this reason, we add the further constraint set 3 of *step constraints*. These ensure that for any two concurrently executed actions $a^- \prec a$, the action that is greater w.r.t. the total order does not depend on any values written by the other action, and thus parallel execution leads to the same global configuration as executing first a^- and then a.

Three Alternative Encodings. We present three different symbolic encodings, namely an *interleaving* encoding, a *static* ∃*-step* encoding and a *dynamic* ∃*-step* encoding. All three encodings contain all unit steps in the sense that if $\langle c, a, c' \rangle \in \Delta^B$ and the valuations of state and next-state variables represent c and c' respectively, then there is a valuation of input variables such that the constraints are satisfied. Conversely, nothing but ∃-steps that respect the order $\prec$ are allowed by the encodings. Let $M^B_\exists = \langle C^B, c_{\text{init}}, \Delta^B_\exists \rangle$ be the state space obtained from M^B by the definition of ∃-step semantics with the further restriction that the total ordering of the set of actions S in each step (the ordering in item 2 of Definition 1) respects the fixed order $\prec$. If the valuation of state variables represents a bounded configuration $c \in C^B$ and the constraints are satisfied, then the valuation of next-state variables represents a bounded configuration $c' \in C^B$ and there is a set of actions $S \subseteq A$ such that the step $\langle c, S, c' \rangle$ belongs to $\Delta^B_\exists$. Furthermore, in the interleaving encoding, S only contains one action, and thus every step is a unit step.



(a) Global configuration (b) State machine of class Node

Fig. 2. An example for illustrating static and dynamic ∃-steps

The definitions of the three encodings overlap for the most part, and the differences are stated explicitly. The difference between static and dynamic steps is that in static steps, whether actions a_1 and a_2 can be executed concurrently depends only on a_1 and a_2. In dynamic steps, it also depends on the current global configuration, in particular, on the values of attributes. Consider the global configuration in Fig. 2(a) consisting of three objects of class Node. Transition t_3 is enabled both in object o_{11} and in o_{12}. Because the action $\langle o_{11}, t_3 \rangle$ increments attribute data in o_{12} and $\langle o_{12}, t_3 \rangle$ increments data in o_{13}, neither action reads a value written by the other. The dynamic step encoding allows $\langle o_{11}, t_3 \rangle$ and $\langle o_{12}, t_3 \rangle$ to be executed concurrently in this global configuration. However, in some other global configuration the succ attribute in o_{11} and o_{12} might refer to the same object, so the two actions might read and modify the same attribute. As a safe statically computed approximation, the static step encoding never allows executing $\langle o_{11}, t_3 \rangle$ and $\langle o_{12}, t_3 \rangle$ concurrently.

3.1 State Variables

The set of state variables contains three kinds of elements for each object o.

1. $\mathsf{Active}(o, s)$, where s is a state in the state machine of o, is true iff s is the current active state in o.
2. $\mathsf{AttrVal}(o, x)$, where x is an attribute of the class of o, determines the current value of $o.x$ and has the same domain as the type of x.
3. $\mathsf{InputQ}(o)$ with domain $Msgs^0 \cup \cdots \cup Msgs^{\text{QSIZE}}$ determines the contents of the input queue of o.

Given a bounded global configuration, the values of state variables can be derived in the obvious way. The corresponding next-state variables are denoted by $next\,(\mathsf{Active}(o, s))$, $next\,(\mathsf{AttrVal}(o, x))$, and $next\,(\mathsf{InputQ}(o))$, respectively.

3.2 State Machines and Queues

This section gives a rough overview of the encoding of state machine control logic. A more detailed definition can be found in [16].

The control logic constraints are responsible for ensuring that in the context of a single object $o \in O$, (i) at most one transition instance or implicit consumption action is executed in one step, (ii) an action is executed only if it is enabled in

the global configuration represented by the state variables, and (iii) the variables *next* (Active(o, s)) correctly reflect the active state after the step.

Let $o \in O$ be an object. The input variable Dispatch(o) with domain $Sigs \cup \{\textbf{none}\}$ determines which message $sig[\ldots]$, if any, is being consumed by o. For each transition t in the state machine of o, the input variable Fire(o, t) determines whether t is being fired in o. We define the actions $S \subseteq A$ of the current step as the set consisting of all transition instances $\langle o, t\rangle$ such that Fire(o, t) is true, and all implicit consumption actions $\langle o, \text{IMPCONS}\rangle$ such that Fire(o, t) is false for all t but Dispatch$(o) \neq$ **none**.

For example in the state machine of Fig. 1(c), the next-state variable related to state showAvgBps is fixed by the constraint *next* (Active$(o,$ showAvgBps$)) \Leftrightarrow$ Fire$(o, t_8) \vee \big(\neg$Fire$(o, t_9) \wedge$ Active$(o,$ showAvgBps$)\big)$, and the enabledness check for the action $\langle o, t_8\rangle$ is encoded in the constraint Fire$(o, t_8) \Leftrightarrow \big($(Dispatch$(o) =$ fbutton$) \wedge$ Active$(o,$ showBps$)\big)$.

Object o is *scheduled* if it is consuming a signal or firing a spontaneously triggered transition, formalized in the function definition

$$\text{Scheduled}(o) := \big(\text{Dispatch}(o) \neq \textbf{none}\big). \tag{1}$$

To rule out an empty step with no actions, we require that at least one object is scheduled in each step by the constraint

$$\bigvee \big\{\text{Scheduled}(o) \mid o \in O\big\}. \tag{2}$$

Furthermore, there are queue constraints ensuring that consumed messages are removed from the front of the input queue and received messages are added to the back of the queue. The input variable NewMsg(o) with domain $Msgs \cup \{\textbf{none}\}$ denotes the message possibly being sent to o. Queue overflows are prevented by specialized constraints, disallowing transitions into global configurations that are not in C^B.

3.3 Effects and Data

Expressions. All data manipulation in the effects of transitions is based on evaluating expressions. We define a function Eval$(expr)$ that gives the value of expression *expr*. For clarity, we leave out the context in which the expression is evaluated; a more rigorous treatment can be found in [16]. Evaluation of constant and infix expressions is straightforward, given the encodings for all infix operators. In the context of an object o, the value of Eval(`this`) is o. Attribute access expressions of the form $refexpr\,.\,\hat{x}$ are encoded as case switches over all objects of the type of *refexpr*. For example, the expression succ.data in Fig. 2(b) translates in the context of o_{11} to the formula

$$\text{Eval(succ.data)} := \mathit{if}\ \begin{cases} \text{AttrVal}(o_{11}, \text{succ}) = o_{11} & : \ \text{AttrVal}(o_{11}, \text{data}) \\ \text{AttrVal}(o_{11}, \text{succ}) = o_{12} & : \ \text{AttrVal}(o_{12}, \text{data}) \\ \text{AttrVal}(o_{11}, \text{succ}) = o_{13} & : \ \text{AttrVal}(o_{13}, \text{data}) \\ \mathit{else} & : \ 0. \end{cases}$$

Effects. The effects of transitions can modify the global configuration by assigning values to attributes and by sending messages to objects. For each transition instance $\langle o, t\rangle$, each object $\hat{o}$, and each attribute $\hat{x}$ of the class of $\hat{o}$, we define functions $\mathsf{Send}_{o,t}(\hat{o})$ and $\mathsf{Write}_{o,t}(\hat{o}, \hat{x})$ that evaluate to true if the transition instance is executed and sends a message to $\hat{o}$ or assigns to $\hat{o}.\hat{x}$, respectively. Assignment can occur explicitly by an assignment statement or implicitly by message argument reception.

These functions are used for determining the next global configuration in the following way. If the effect of a transition t in the state machine of a class C contains a send statement `send` sig`(`$arg_1, \ldots, arg_m$`) to` $targetexpr$`;`, we add for each object o of class C and each object $\hat{o}$ of the type of $targetexpr$ the constraint

$$\mathsf{Send}_{o,t}(\hat{o}) \Rightarrow \big(\mathsf{NewMsg}(\hat{o}) = sig[\mathsf{Eval}(arg_1), \ldots, \mathsf{Eval}(arg_m)]\big), \tag{3}$$

which fixes the message received by $\hat{o}$ in case $\langle o, t\rangle$ is executed. Similarly, the value assigned to $\hat{o}.\hat{x}$ by a transition instance $\langle o, t\rangle$ is fixed by the constraint

$$\mathsf{Write}_{o,t}(\hat{o}, \hat{x}) \Rightarrow \big(next\,(\mathsf{AttrVal}(\hat{o}, \hat{x})) = \mathsf{Temp}_{o,t}(\hat{o}, \hat{x})\big), \tag{4}$$

where $\mathsf{Temp}_{o,t}(\hat{o}, \hat{x})$ evaluates to the value of $\hat{o}.\hat{x}$ after executing the effect of t in object o. This is defined using the Eval function. For example in Fig. 2(b),

$$\mathsf{Temp}_{o_{11},t_3}(o_{12}, \mathsf{data}) := \mathit{if}\ \begin{cases} \mathsf{AttrVal}(o_{11}, \mathsf{succ}) = o_{12} & : \ \mathsf{Eval}(\mathsf{succ.data + 1}) \\ \mathit{else} & : \ \mathsf{AttrVal}(o_{12}, \mathsf{data}). \end{cases}$$

For each transition instance $\langle o, t\rangle$ and each assertion `assert` $condexpr$`;` in the effect of t, we check that the invariant $\mathsf{Fire}(o, t) \Rightarrow \mathsf{Eval}(condexpr)$ holds.

Frame Conditions. Let $\Theta \subseteq A$ be the set of all transition instances. The only situation when $\mathsf{NewMsg}(\hat{o})$ is not fixed by (3) is when $\mathsf{Send}_{o,t}(\hat{o})$ is false for all transition instances $\langle o, t\rangle \in \Theta$. Similarly, $next\,(\mathsf{AttrVal}(\hat{o}, \hat{x}))$ is not fixed when all functions $\mathsf{Write}_{o,t}(\hat{o}, \hat{x})$ are false. To fix these, we add the constraints

$$\neg \bigvee \big\{\mathsf{Send}_{o,t}(\hat{o}) \mid \langle o, t\rangle \in \Theta\big\} \Rightarrow \big(\mathsf{NewMsg}(\hat{o}) = \mathbf{none}\big), \tag{5}$$

$$\neg \bigvee \big\{\mathsf{Write}_{o,t}(\hat{o}, \hat{x}) \mid \langle o, t\rangle \in \Theta\big\} \Rightarrow \big(next\,(\mathsf{AttrVal}(\hat{o}, \hat{x})) = \mathsf{AttrVal}(\hat{o}, \hat{x})\big). \tag{6}$$

3.4 Step Constraints

In the interleaving encoding, at most one object is scheduled at a time, as required by the constraint

$$\mathit{AtMostOne}\,(\{\mathsf{Scheduled}(o) \mid o \in O\})\,. \tag{7}$$

Consequently, at most one action is executed in each step. A predicate of the form $\mathit{AtMostOne}\,(P)$ evaluates to **true** if and only if zero or one of the predicates

in set P evaluates to **true**. This can be expressed with $\mathcal{O}(|P|)$ binary Boolean connectives.

In the $\exists$-step encodings, (7) is replaced by more liberal constraints as follows. We require that an action must not send a message to an object if a preceding action has already done so, and it must not read an attribute that a preceding action has written. This is formalized in the constraints

$$\bigvee \left\{ \mathsf{Send}_{o^-,t^-}(\hat{o}) \mid \langle o^-, t^- \rangle \prec \langle o, t \rangle \right\} \Rightarrow \neg \mathsf{Send}_{o,t}(\hat{o}), \tag{8}$$

$$\bigvee \left\{ \mathsf{Write}_{o^-,t^-}(\hat{o}, \hat{x}) \mid \langle o^-, t^- \rangle \prec \langle o, t \rangle \right\} \Rightarrow \neg \mathsf{Read}_{o,t}(\hat{o}, \hat{x}). \tag{9}$$

In the implementation, we employ maximal sharing of subformulas in the left-hand sides of (5), (6), (8), and (9), resulting in an encoding that is linear instead of quadratic in the number of transition instances. Furthermore, the left-hand sides of (8) and (9) are "free" in the sense that they are already present as subformulas of (5) and (6), and can therefore be shared.

The function $\mathsf{Read}_{o,t}(\hat{o}, \hat{x})$, which appears in constraint (9), is defined so that it evaluates to true if the transition instance $\langle o, t \rangle$ is executed and reads the attribute $\hat{o}.\hat{x}$. The constraint says that a transition instance $\langle o, t \rangle$ that reads an attribute $\hat{o}.\hat{x}$ can only be executed if that attribute is not modified by any concurrently executed transition instance that precedes $\langle o, t \rangle$ in the total order. This means that in the global configuration that would result from executing the preceding transition instances, the value of $\hat{o}.\hat{x}$ is still the same as in the starting configuration represented by the state variables. This justifies the use of $\mathsf{AttrVal}(\hat{o}, \hat{x})$ in evaluating the expressions in the effect of $\langle o, t \rangle$. Also notice that (4) forbids executing two transition instances that would assign a different value to the same attribute, and (8) prevents two transition instances from sending to the same object.

Implicit consumption actions cannot send messages or modify attributes. However, an implicit consumption action $\langle o, \text{IMPCONS} \rangle$ can implicitly *read* an attribute because the enabledness of $\langle o, \text{IMPCONS} \rangle$ might depend on the enabledness of a transition instance $\langle o, t \rangle$, which in turn might depend on an attribute that is mentioned in the guard of t. By setting the requirement that implicit consumption actions precede all transition instances in the total order, we rule out the possibility of $\langle o, \text{IMPCONS} \rangle$ implicitly reading an attribute that has been written by a preceding action. No extra constraints are thus needed.

In order to strictly confine the analysis to the bounded transition relation $\Delta^B_\exists$, additional step constraints are placed that prevent a queue from growing past its bound and shrinking back within a single step. The details are in [16].

Static and Dynamic Steps. The difference between the two $\exists$-step encodings is in the definitions of Send, Write, and Read. In dynamic steps, these functions are evaluated accurately using the input and state variables. For example, $\mathsf{Send}_{o,t}(\hat{o})$ is defined as $\mathsf{Fire}(o, t) \wedge \big(\mathsf{Eval}(\mathit{targetexpr}) = \hat{o}\big)$, and $\mathsf{Read}_{o,t}(\hat{o}, \hat{x})$ is true iff $\mathsf{Fire}(o, t)$ is true and $\mathsf{Eval}(\mathit{refexpr}) = \hat{o}$ is true for any subexpression of the form $\mathit{refexpr}.\hat{x}$ in the guard or effect of t. The same definitions are used in the interleaving encoding.

In static steps, overapproximations are used. If the guard or effect of transition t does not contain $\hat{x}$ in any subexpression, then $\mathsf{Read}_{o,t}(\hat{o}, \hat{x})$ is trivially **false** for all o and $\hat{o}$. Otherwise, $\mathsf{Read}_{o,t}(\hat{o}, \hat{x})$ is defined as $\mathsf{Fire}(o, t)$. Conceptually this means that when $\langle o, t\rangle$ is executed, it reads the attribute $\hat{x}$ of *all* objects that contain it. Equivalently, $\mathsf{Send}_{o,t}(\hat{o})$ is defined as $\mathsf{Fire}(o, t)$ if the effect of t contains any send statement to an object of the same class as $\hat{o}$ and **false** otherwise, and similarly for Write. This approximation strengthens the step constraints (8) and (9) and also makes the constraints static in the sense that they no longer refer to the state variables. As an optimization, if transition t only accesses $\hat{x}$ using the expression `this`.$\hat{x}$, then it is known that the action $\langle o, t\rangle$ does not read $\hat{o}.\hat{x}$ if $\hat{o} \neq o$, and therefore $\mathsf{Read}_{o,t}(\hat{o}, \hat{x})$ is defined as **false** in these cases. The same optimization is applied to Send and Write.

Because the function $\mathsf{Send}_{o,t}(\hat{o})$ is an approximation in the static step encoding, it may evaluate to **true** even though no message is being sent to $\hat{o}$. For this reason, in static steps we replace (3) with two constraints

$$\mathsf{Send}_{o,t}(\hat{o}) \wedge \big(\mathsf{Eval}(\mathit{targetexpr}) = \hat{o}\big) \Rightarrow \big(\mathsf{NewMsg}(\hat{o}) = \mathit{sig}[\ldots]\big), \tag{10}$$

$$\mathsf{Send}_{o,t}(\hat{o}) \wedge \neg\big(\mathsf{Eval}(\mathit{targetexpr}) = \hat{o}\big) \Rightarrow \big(\mathsf{NewMsg}(\hat{o}) = \mathbf{none}\big). \tag{11}$$

A similar correction does not need to be made to (4) because $\mathsf{Temp}_{o,t}(\hat{o}, \hat{x})$ is evaluated accurately even in the static step encoding.

Consider again the setting of Fig. 2. Assuming that $\langle o_{11}, t_3\rangle \prec \langle o_{12}, t_3\rangle$ are the two first transition instances in the total order, we get from (9) the step constraint

$$\mathsf{Write}_{o_{11},t_3}(\hat{o}, \mathsf{data}) \Rightarrow \neg\mathsf{Read}_{o_{12},t_3}(\hat{o}, \mathsf{data}) \tag{12}$$

instantiated for each $\hat{o} \in \{o_{11}, o_{12}, o_{13}\}$. In dynamic steps, these expand to

$$\mathsf{Fire}(o_{11}, t_3) \wedge \big(\mathsf{AttrVal}(o_{11}, \mathsf{succ}) = \hat{o}\big) \Rightarrow \neg\big(\mathsf{Fire}(o_{12}, t_3) \wedge (\mathsf{AttrVal}(o_{12}, \mathsf{succ}) = \hat{o})\big),$$

allowing, for example, executing $\langle o_{11}, t_3\rangle$ and $\langle o_{12}, t_3\rangle$ concurrently in the global configuration of Fig. 2(a). In static steps, all three instantiations of (12) reduce to the same constraint $\mathsf{Fire}(o_{11}, t_3) \Rightarrow \neg\mathsf{Fire}(o_{12}, t_3)$, which disallows concurrent execution of the two actions in any global configuration. In this example, static $\exists$-step semantics yields a smaller encoding, but dynamic $\exists$-step semantics permits more concurrency in a single step.

3.5 Size of the Encodings

Let $|M|$ be the size of the model, containing the definition of every class, attribute, signal, and state machine, and the textual definitions of guards and effects. Assuming that common subformulas are shared between constraints, the size of all three encodings is $\mathcal{O}(|M|(\textsc{qsize} \cdot |O| + |O|^2) \log |O|)$. The term $\log |O|$ is the required number of bits to represent the values of attributes and expressions of reference type. The term $|O|^2$ appears because objects can refer to each other arbitrarily, and it seems unavoidable in the presence of dynamic references. The total size of queue and state machine encodings without data or transition effects is $\mathcal{O}(|M| \cdot \textsc{qsize} \cdot |O| \log |O|)$.

The worst-case size for the data and step encoding can be seen in (4). For each assignment statement (quantity bounded by $|M|$), there are $\mathcal{O}(|O|^2)$ instantiations of (4), each of size $\mathcal{O}(\log |O|)$ because of the comparison of object references. Thus the total size sums up to $\mathcal{O}(|M| \cdot |O|^2 \log |O|)$ even in the interleaving encoding. In the $\exists$-step encodings, there are $\mathcal{O}(|M| \cdot |O|^2)$ additional step constraints, but they do not dominate the total encoding size in the experiments.

4 Experimental Results

We have implemented the symbolic encoding described above.[1] The tool assumes the system models to be described with a subset of UML, the main additions to the above encoding being that state machines also support (i) hierarchy, (ii) completion events via the busy-quiescent construction given in [5], (iii) deferring of events, and (iv) initial and choice pseudostates (see [19] for a symbolic *interleaving* semantics encoding of such extended state machines). It outputs the encoding as a NuSMV [1] program and currently supports model checking queries for deadlocks, implicit consumption of messages, assertion violations, and action language run-time errors. The tool chooses the fixed ordering of actions arbitrarily but deterministically based on the identifiers in the input file. The following experiments were run on a PC machine with a 2 GHz AMD Athlon 64 processor, 2 GB of memory, and Debian Linux operating system. We used version 2.4.3 of NuSMV and limited the available memory to 1.5 GB and time to ten minutes. The width of integer attributes in the encoding was 32 bits and the input queue size was two.

We used the following models. (i) SCP is a simple communication protocol with three active objects (environment, sender, and receiver) and three cycling phases (connection establishment, data transfer, connection release). (ii) travel is an essentially sequential resource allocation process modeling a travel agent accessing a database, involving 4 active objects. (iii) mtravel is a variant of travel with 3 competing travel agents and databases organized in a ring. (iv) giop1_2 has been adapted from the General Inter-ORB Protocol model [20], with an uninitialized variable introduced in the adaptation.

Table 1 shows the results. The column "sem." gives the semantics (interleaving, static $\exists$-steps, or dynamic $\exists$-steps) and the smallest bound for a counterexample under it, "SBMC zchaff" gives the minimum and maximum running time (in seconds) of 10 runs of the incremental BMC algorithm [21,22] when ZChaff is used as the SAT solver, "SBMC minisat" is the same with MiniSat as the SAT solver, and "BDD invar" gives the running times of a BDD-based invariant checking. The numbers in square brackets are the times used by the SAT solvers instead of the total running times of NuSMV (especially on the model giop1_2, the total running time is dominated by preprocessing overhead). M.O. means that all runs exceeded the memory limit, and T.O.(x–y) means that all runs

[1] The source code and the models used in the experiments are available at `http://www.tcs.hut.fi/Research/Logic/SMUML.shtml`

Table 1. Results

model + property	sem.	k	SBMC zchaff time	SBMC minisat time	BDD invar time	Spin DFS time	Spin DFS |cex|	Spin DFS -i time	Spin DFS -i |cex|
travel	interl.	15	0.59–0.64	0.46–0.55	1.19–1.20	0.01–0.01	17–24	0.03–0.05	15–15
deadlock	s.step	11	0.32–0.36	0.30–0.34	1.91–1.93				
	d.step	11	0.31–0.35	0.31–0.34	1.69–1.71				
mtravel	interl.	36	T.O.(21–22)	T.O.(23–24)	M.O.	0.01–43.58	71–103576	T.O.	
deadlock	s.step	14	5.87–10.73	3.26–5.03	M.O.				
	d.step	11	2.01–2.26	1.56–1.67	M.O.				
SCP	interl.	13	2.17–2.70	1.21–1.51	174.05–174.89	0.02–0.03	13–104	0.04–0.74	13–13
deadlock	s.step	7	0.37–0.41	0.37–0.40	107.04–107.42				
	d.step	6	0.38–0.40	0.37–0.40	T.O.				
SCP	interl.	7	0.46–0.49	0.42–0.45	n.a.	0.01–0.01	12–24	0.02–0.04	7–7
implicit	s.step	6	0.38–0.41	0.37–0.41	n.a.				
cons.	d.step	5	0.37–0.39	0.36–0.40	n.a.				
giop1_2	interl.	14	293.09–338.47 [79.06–124.39]	250.15–261.90 [36.30–48.49]	n.a.	0.29–580.96	30–64	T.O.	
runtime errors	s.step	9	162.59–174.98 [6.71–9.73]	156.94–165.81 [2.16–3.34]	n.a.				
	d.step	8	156.57–162.53 [4.04–5.13]	154.23–158.20 [1.19–2.30]	n.a.				

timed out; x and y give the minimum and maximum, respectively, of the bounds that were reached before timeout. We check (i) deadlocks of the models SCP, mtravel, and travel, (ii) whether implicit consumption of messages is possible in the model SCP, and (iii) if the model giop1_2 has run-time errors. The latter two properties are only checked with BMC but not with BDDs as the invariant involves input variables; this is not accepted by the NuSMV BDD invariant checking command.

Analyzing the sizes of SAT instances generated by NuSMV shows that the proportion of $\exists$-step constraints in an instance is 4–8 % when static steps are used, and 5–15 % when dynamic steps are used, depending on the model. This verifies the presumption that using step semantics does not substantially increase the size of the encoding.

From the results we see that using $\exists$-step semantics instead of interleaving can (i) drop the bound required to find a counterexample, and (ii) more importantly, quite radically reduce the running times of BMC algorithms. This is especially true with models that contain lots of concurrency. E.g. on the model mtravel using interleaving semantics, BMC could not reach the required bound 36 even if we gave 1 hour of time, while using step semantics a corresponding counterexample is found within seconds. With BDDs, it seems in fact that the interleaving semantics is quite competitive with the step semantics. This is an interesting finding requiring further study. We also experimented with the BDD-based breadth-first enumeration of reachable states in NuSMV and the findings were that the step semantics indeed covered more states than the interleaving semantics with the same number of iterations but it took more time to do so.

We also ran tests with the state-of-the-art explicit state model checker Spin [2] that is designed especially for the analysis of this kind of models. The UML models were automatically translated to the input language of Spin by a translation based on that in [5]. The last two columns in Table 1 give the running times and lengths of produced counterexamples of Spin with (i) the default depth-first search mode

("Spin DFS"), and (ii) the same with counterexample minimization option "-i" enabled. Partial order reductions and state compression ("COLLAPSE") were enabled in both modes. Different runs on the same model produced diverse results as the order of variable and process declarations in the Spin model varied between runs of the translator program due to some non-determinism in the libraries used. As expected, Spin is superior on models with relatively small state spaces (SCP and travel). On the models mtravel and giop1_2 containing more concurrency, Spin sometimes consumes more time. More importantly, in cases it manages to find a counterexample, the produced counterexamples are often much longer than the minimal ones produced by BMC based methods. These very long counterexamples are not as useful in debugging the system as it becomes much harder for the user to locate the real source of the bug.

5 Conclusions and Future Work

We have shown how to exploit the concurrency in the transition relation encoding for object based communicating state machines. Especially in bounded model checking, the proposed ∃-step semantics significantly outperform the traditional interleaving semantics approach, without any considerable blowup in the encoding as a SAT formula.

Our experimental results show that when searching for short counterexamples, symbolic model checking, especially with ∃-step semantics, can provide a competitive approach to model checking of asynchronous message passing protocols. This has traditionally believed to be a field where only explicit state model checkers can be efficient. We show that by using bounded model checking with our step encodings, we can find much shorter counterexamples than Spin does, and achieve this with competitive running times.

One avenue for further study is the use of SAT modulo theories (SMT) solvers to improve the performance of bounded model checking of systems containing data. Our encoding can be fairly easily adjusted to do that. Also requiring further study are the details of the way the ∃-step semantics needs to be restricted in order to soundly accommodate the model checking of liveness properties along the lines of [23,22]. And as the choice of the total ordering between actions in the encoding affects which steps are considered, a statically chosen good ordering might improve performance, but this needs further investigation.

References

1. 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, pp. 359–364. Springer, Heidelberg (2002)
2. Holzmann, G.J.: The Spin Model Checker. Addison-Wesley, Reading (2004)
3. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) ETAPS 1999 and TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)

4. Valmari, A.: The state explosion problem. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
5. Jussila, T., Dubrovin, J., Junttila, T., Latvala, T., Porres, I.: Model checking dynamic and hierarchical UML state machines. In: Proc. MoDeV^{2}a: Model Development, Validation and Verification, pp. 94–110 (2006)
6. Rintanen, J., Heljanko, K., Niemelä, I.: Planning as satisfiability: parallel plans and algorithms for plan search. Artificial Intelligence 170(12-13), 1031–1080 (2006)
7. Heljanko, K.: Bounded reachability checking with process semantics. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 218–232. Springer, Heidelberg (2001)
8. Best, E., Devillers, R.R.: Sequential and concurrent behaviour in Petri net theory. Theoretical Computer Science 55(1), 87–136 (1987)
9. Kautz, H.A., Selman, B.: Pushing the envelope: Planning, propositional logic and stochastic search. In: AAAI 1996/IAAI 1996, vol. 2, pp. 1194–1201. AAAI Press, Menlo Park (1996)
10. Dimopoulos, Y., Nebel, B., Koehler, J.: Encoding planning problems in nonmonotonic logic programs. In: Steel, S. (ed.) ECP 1997. LNCS, vol. 1348, pp. 169–181. Springer, Heidelberg (1997)
11. Wehrle, M., Rintanen, J.: Planning as satisfiability with relaxed ∃-step plans. In: Orgun, M.A., Thornton, J. (eds.) AI 2007. LNCS (LNAI), vol. 4830, pp. 244–253. Springer, Heidelberg (2007)
12. Ogata, S., Tsuchiya, T., Kikuno, T.: SAT-based verification of safe Petri nets. In: Wang, F. (ed.) ATVA 2004. LNCS, vol. 3299, pp. 79–92. Springer, Heidelberg (2004)
13. Jussila, T.: BMC via dynamic atomicity analysis. In: ACSD 2004, pp. 197–206. IEEE Computer Society, Los Alamitos (2004)
14. Jussila, T., Heljanko, K., Niemelä, I.: BMC via on-the-fly determinization. International Journal on Software Tools for Technology Transfer 7(2), 89–101 (2005)
15. Jussila, T.: On Bounded Model Checking of Asynchronous Systems. Doctoral dissertation, Helsinki Univ.of Technology (2005)
16. Dubrovin, J., Junttila, T., Heljanko, K.: Symbolic step encodings for object based communicating state machines. Technical Report B24, Helsinki Univ.of Technology, Lab.for Theoretical Computer Science (2007)
17. Dubrovin, J.: Jumbala — An action language for UML state machines. Research Report A101, Helsinki Univ.of Technology, Lab.for Theoretical Computer Science (2006)
18. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley, Reading (2005)
19. Dubrovin, J., Junttila, T.: Symbolic model checking of hierarchical UML state machines. In: ACSD (to appear, 2008)
20. Kamel, M., Leue, S.: Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. International Journal on Software Tools for Technology Transfer 2(4), 394–409 (2000)
21. Heljanko, K., Junttila, T., Latvala, T.: Incremental and complete bounded model checking for full PLTL. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 98–111. Springer, Heidelberg (2005)
22. Biere, A., Heljanko, K., Junttila, T., Latvala, T., Schuppan, V.: Linear encodings of bounded LTL model checking. Logical Methods in Computer Science 2(5:5) (2006)
23. Heljanko, K., Niemelä, I.: Bounded LTL model checking with stable models. Theory and Practice of Logic Programming 3(4&5), 519–550 (2003)

Modeling and Model Checking Software Product Lines

Alexander Gruler*, Martin Leucker, and Kathrin Scheidemann

Institut für Informatik, Technische Universität München, Germany

Abstract. Software product line engineering combines the individual developments of systems to the development of a family of systems consisting of common and variable assets.In this paper we introduce the process algebra PL-CCS as a product line extension of CCS and show how to model the overall behavior of an entire family within PL-CCS. PL-CCS models incorporate behavioral variability and allow the derivation of individual systems in a systematic way due to a semantics given in terms of multi-valued modal Kripke structures. Furthermore, we introduce multi-valued modal μ-calculus as a property specification language for system families specified in PL-CCS and show how model checking techniques operate on such structures. In our setting the result of model checking is no longer a simple *yes* or *no* answer but the set of systems of the product line that do meet the specified properties.

1 Introduction

A *software product line* is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets [CN02]. Developing a set of related systems as a product line, rather than every system individually, enables the systematic exploitation of synergy effects. Product line concepts are, for example, widely used in automotive production, in order to allow for individuality and high customizability of vehicles, while at the same time keeping production cost low.

Modern vehicles are controlled by large sets of configurable electronic control units (ECUs) which communicate via buses and gateways. In other words, they are highly individual, distributed, embedded systems on wheels. Due to the prevalence of new technologies, the complexity of vehicle systems will increase further. In order to be able to guarantee high quality, security and safety requirements in future, the application of formal verification and analysis techniques get more and more essential. However, in order to be effectively applied in a product line context, formal modeling approaches as well as the applied formal verification techniques, need to efficiently cope with variability.

Despite their importance for the development of software intensive systems, formal modeling and especially verification techniques for product lines do not yet meet the industrial needs.

Kishi et al. in [KNK05] proposed to use traceability in a product model in order to automatically compose subsystem models of a configuration in order to model check

* This author was partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of project VEIA under grant 01ISF15A.

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 113–131, 2008.

that configuration. With this approach it is possible to automate the composition of configurations, but it is still necessary to model check a model for each configuration separately.

Modular verification approaches like the one introduced by Li, Krishnamurthi and Fisler [LKF05] use compositional verification techniques in order to infer properties of an assembled system from the properties of its assets. Interfaces of feature-oriented modules contain constraints, similar to verification conditions that other modules must satisfy at composition time. In their approach, these conditions are automatically derived during feature verification. Configurations are verified individually, but results for partly integrated configurations and modules, which have already been verified, can of course be reused.

Larsen et al. defined a behavioral variability model for product line development based on modal I/O automata [LNW07], which are an extension of Larsen's and Thomsen's Kripke modal transition systems [LT91]. Their aim is not to verify product specific functional properties for configurations, but rather to verify the error free combinability of interfaces. Error free composition is characterized by the absence of deadlocks. It is not required that all possible configurations give an error free composition, but only that there exist configurations that can avoid errors under suitable use.

Kripke modal transition systems are used in [FUB07] to study the notion of *behavioral conformance* in the setting of software product lines.

In our approach we model the functional behavior of an entire product family in a single model which explicitly incorporates behavioral variability. Compared to other approaches which also contain the concept of variability in arbitrary development assets such as requirement specifications, design models or test models [PBvdL05], we have included the variability information into the behavioral model. In particular, the concept of variability is also considered in the semantical model. In contrast to other techniques, our model and the respective semantics allow the application of model checking techniques. By this, we can reduce typical questions from product-line engineering to model checking problems.

More specifically, this paper introduces PL-CCS as a variant of Milner's CCS [Mil95] designed to model the interaction of software components used in software product lines. While Milner's CCS is well-suited for describing the communication of (closed) software systems, it lacks support for defining a set of systems. We extend CCS by a variants operator $\oplus$, which allows to model alternative behavior, i.e. alternative processes, with the meaning that only one of the alternative processes will be existing in the final (running) system. The semantics of an PL-CCS system is defined in terms of a *labeled transition system for product lines* (PL-LTS), which is essentially a multi-modal Kripke structure extending Kripke modal transition systems [LT91]. Abstracting from the specification concept on top (in our case extended CCS), a PL-LTS assigns a semantics to the so far only vaguely defined notion of variability. Note that de Nicola et. al. [VN98] and Majster-Cederbaum [MC01] introduce as well an $\oplus$-operator which represents a form a variability. However, the approach is not tailored to product lines and does not study the question of verification.

The main benefit of the proposed modeling formalism is that it caters for automatic verification by model checking. We introduce the multi-valued modal μ-calculus as

combination of Kozen's modal μ-calculus [Koz83] and multi-valued μ-calculus as defined by Grumberg and Shoham [SG05], yielding a property specification language suitable for specifying and checking properties of PL-CCS programs. More specifically, the result of model checking a property for a PL-CCS program is the *set* of configurations satisfying the property at hand and not only the answer if the property holds or not.

2 Product-Line CCS

In this section we introduce PL-CCS as an extension of Milner's process algebra CCS [Mil95]. PL-CCS is designed for modeling the behavior of an entire product line in a way especially suitable for automatic verification by model checking.

2.1 Product Lines

Before giving a formal approach to our notion of product lines, let us consider an example, and derive our formalism in an intuitive manner. Hereby, we will also fix the terminology we use in the rest of this paper, mostly following [CN02], where a *product line* is considered to be a set of systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets.

As usual, we consider individual systems (products) that are built from subsystems in a compositional manner. Following CCS, we model the behavior of a system as well as its subsystems as processes. The parallel operator $\|$ known from CCS can be used to express that the behavior of the compound system is defined by the parallel execution of its subsystems.

For example, a car C consists—amongst other things—of an engine E, a locking system LS, and an infotainment system IS, which operate in parallel. This is denoted as:

$$C \stackrel{def}{=} E \parallel LS \parallel IS$$

In the product lines we consider, subsystems may be realized by alternative *variants*. Entire subsystems may even be optional. For example, vehicles may be equipped with different locking systems, such as (i) a central locking system $LS_{central}$ controlling the locking of all doors, or alternatively (ii) a locking system $LS_{keyless}$, which allows remote keyless entry via a key fob. Such variants can be specified in PL-CCS using the binary *variants operator* $\oplus$. The usage of the variants operator can be understood as offering a set of possible "choices" realizing a variant. Thus, the *locking system* can be specified as follows:

$$LS \stackrel{def}{=} LS_{central} \oplus LS_{keyless}$$

As not all vehicles in the specified product line may be equipped with an infotainment system, we enrich PL-CCS with an *optional operator* $\langle \_ \rangle$, allowing the infotainment system to be declared as *optional*, written as:

$$C \stackrel{def}{=} E \parallel LS \parallel \langle IS \rangle$$

Both, the variants operator and the optional operator define a *variation point*. Given a PL-CCS model for an entire product line, individual systems can be derived by making

decisions for all variation points. More precisely, choosing for every variants operator one variant and for every optional operator whether the optional subsystem is present or not, yields a specific *configuration*. The configuration then defines uniquely one *system*, also called *product*, that is derivable from the product line.

Note that CCS offers an operator $+$ for expressing *non-deterministic choice*. Although our variants operator $\oplus$ is to some extent similar to the CCS $+$, there are several important conceptual as well as formal differences between both. Thus, both operators are essential for PL-CCS (see also Rules 12 and 13 in Section 2.3).

2.2 PL-CCS – Syntax

In this section we introduce the syntax of PL-CCS programs, which allows us to define design models for software product lines.

Let Id be a finite set of *process identifiers* and Σ by a finite set of *input actions*. Usually, $P, Q, P_1, \ldots$ range over process identifiers and $a, b, \ldots$ range over input actions. As in CCS, let $\mathcal{A} = \Sigma \cup \bar{\Sigma} \cup \{\tau\}$ represent the set of *communication actions*, where $\tau \notin \Sigma \cup \bar{\Sigma}$ represents the *silent action*, and, $\bar{\Sigma} = \{\bar{a} \mid a \in \Sigma\}$ is the set of *output actions*. Usually, $\alpha, \beta, \ldots$ range over communication actions. By Nil, we denote the atomic *idle process.*

The set $\mathcal{P}$ of all *PL-CCS process expressions* (or short processes) is generated by the following grammar:

$$e ::= Q \mid Nil \mid \alpha.e \mid e + e \mid e \oplus e \mid e \parallel e \mid e[f] \mid e\backslash L \tag{1}$$

where $Q \in Id$ is a process identifier, $\alpha \in \mathcal{A}$ is an action, $L \subseteq \mathcal{A}$ is a set of action labels, and $f : \mathcal{A} \mapsto \mathcal{A}$ is a *renaming function*, i. e. a function respecting $f(\bar{a}) = \overline{f(a)}$ and $f(\tau) = \tau$.

Thus, syntactically, PL-CCS extends CCS [Mil95] only by the binary *variants operator* $\oplus$. An optional-operator $\langle \_ \rangle$ can be added to PL-CCS as the syntactical abbreviation:

$$\langle P \rangle := P \oplus Nil$$

In Section 2.3, where PL-CCS semantics is discussed, we will see that this abbreviation meets our intuition, allowing us to confine in this technical presentation of PL-CCS to the variants operator $\oplus$ only.

A *process definition* is an equation of the form $P \stackrel{def}{=} e$, where $P \in Id$ is a process identifier and $e \in \mathcal{P}$ is a PL-CCS process. We specify the behavior of an entire product family by a *PL-CCS program*: A *PL-CCS program* $Prog$ is a tuple $(\mathcal{E}, P_1)$, where $\mathcal{E}$ is a finite set of *process definitions* and $P_1 \in Id$ is the distinguished *main process identifier* of $Prog$. Typically, we denote a PL-CCS program by listing its equations, assuming that the left-hand side of the first equation is the main process identifier. Thus, we usually write only the set of defining equations as shown aside.

$$\begin{aligned} P_1 &\stackrel{def}{=} e_1 \\ P_2 &\stackrel{def}{=} e_2 \\ &\vdots \\ P_n &\stackrel{def}{=} e_n \end{aligned}$$

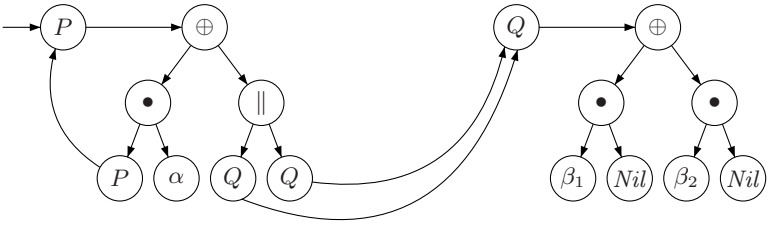


Fig. 1. A program dependency graph

Well-formed PL-CCS programs. Our goal is to model software product lines which require only an *a priori finite* number of decisions taken at variation points when deriving a specific system, which is the case for all product lines relevant in practice. So far, however, as in CCS, PL-CCS allows the creation of new processes by using the parallel operator $\|$ within recursive process definitions. In combination with our $\oplus$-operator this may potentially result in an unbounded number of variation points.

To avoid this, we consider only a (syntactically) restricted subset of all PL-CCS programs. The syntactical restriction is achieved by three conditions: *completeness*, *finitely configurable*, and *fully expanded*, that are used to derive the notion of *well-formed* systems. For such well-formed PL-CCS programs we can define a *compositional* semantics (see section 2.3) which is exactly what we require from a product line approach. In the remainder of the section we successively introduce the syntactical restriction.

Definition 1 (complete PL-CCS program). *We call a PL-CCS program with the set of process definitions* $\{P_1 \stackrel{def}{=} e_1, \ldots, P_n \stackrel{def}{=} e_n\}$ complete*, if all process identifiers* P_i *on the left-hand sides of the defining equations are pairwise distinct and the defining equations* $e_1, \ldots, e_n$ *contain only process identifiers in the set* $\{P_1, \ldots, P_n\}$.

In the following, we consider only complete PL-CCS programs. Now, we turn towards the definition of a dependency graph of a PL-CCS program, which—similar to a control flow graph for programming languages—reflects the dependencies of process definitions in a program.

For a PL-CCS process e, let $pt(e)$ denote the parse tree of e defined in the usual manner as a tree labeled with operator symbols or process identifiers (in leafs). Given a complete PL-CCS program $\big(\{P_1 \stackrel{def}{=} e_1, \ldots, P_n \stackrel{def}{=} e_n\},\ P_1\big)$, we define its *program dependency graph* as the directed labeled graph given as follows: Its nodes comprise those for left-hand sides of the equations, labeled $P_1, \ldots, P_n$, together with the nodes of the parse trees for the right-hand sides of the equations. Its edges comprise the edges of the parse trees plus edges connecting left-hand sides of equations P_i to the roots of parse trees of the corresponding right-hand sides e_i. Additionally, we add edges from leafs of the parse trees labeled P_i to the node for the left-hand side of equation $P_i = e_i$. As an example, consider the following PL-CCS program whose program dependency graph is shown in Figure 1.

$$P \stackrel{def}{=} (\alpha.P) \oplus (Q \parallel Q) \qquad Q \stackrel{def}{=} \beta_1.Nil \oplus \beta_2.Nil$$

We call a node labeled Q *reachable* from a node labeled P if there exists a path from P to Q in its program dependency graph.

Intuitively, a program dependency graph reflects the dependencies between the process identifiers of a PL-CCS program with respect to its defining equations. A cycle in this graph that contains a node labeled by a parallel operator might represent a recursive process definition "spawning" an arbitrary number of copies of its own. If in such a context, the variants operator $\oplus$ comes into play, an unbounded number of configuration selections would be possible. We therefore consider in the following PL-CCS programs which forbid such a situation and thus are configurable within finitely many configuration selections.

Definition 2 (finitely configurable PL-CCS program). *We call a complete PL-CCS program* finitely configurable*, if its program dependency graph has no cycle containing a node labeled with $\|$ from which a node labeled with $\oplus$ is reachable.*

Consider Figure 1. While there is a cycle from P back to P from which a $\oplus$-operator is reachable, the program is finitely configurable as this cycle does not contain a node labeled $\|$. If instead $P \stackrel{def}{=} (\alpha.P) \parallel (Q \parallel Q)$, the program would not be finitely configurable, as the cycle from P to P would contain the parallel operator, and, still the $\oplus$-operator of the second equation is reachable.

Note that the definition of finitely configurable does not *characterize* the programs that are configurable within finitely many configuration selections, but is just a sufficient condition. However, as it is (already) undecidable whether a CCS program yields a finite or infinite state system, it is easy to see that it is also undecidable whether the transition system defined by a PL-CCS program would make use of only finitely many configuration selections. In the following, we therefore consider only on finitely configurable PL-CCS programs.

There is a further restriction we want to make. Consider the two independent systems P and R:

$$\begin{array}{l} P \stackrel{def}{=} Q \parallel Q \\ Q \stackrel{def}{=} Q_1 \oplus Q_2 \end{array} \qquad \text{vs.} \qquad R \stackrel{def}{=} Q_1 \oplus Q_2 \parallel Q_1 \oplus Q_2$$

When considering the left system P one might understand its meaning as follows: (i) P consists of two "instances" of the same variation Q. Hence, one selects once between Q_1 and Q_2 and follows this choice for any occurrence of Q in P. However, if we would specify P by expanding the definition of Q in the definition of P, we would get a system like R, which represents another intention: (ii) In R, we now have two (independent) variation points, which— though offering the same variants Q_1 and Q_2— might be configured differently from each other.

So far, the structural semantics rules, as we introduce them in Section 2.3, are only *compositional* for meaning (ii). Therefore—for the scope of this paper and to simplify the technical treatment—we only consider systems like R, where every variants operator can be configured independently from the configuration of other variation points. Note, that it is easy to extend our formalism to actually cope with both meanings, by introducing a second alternative operator with a suitable semantics for the case of P. However, as this would make the current presentation more technical, we refrain from giving this extension in this paper.

Definition 3 (fully expanded PL-CCS program). *We call a complete and finitely configurable PL-CCS program* fully expanded*, if its program dependency graph satisfies the following: Removing all edges that are part of strongly connected components (yielding a possibly not connected graph), there is at most one path from every node to any $\oplus$ node.*

Note, that a finitely configurable PL-CCS program which is not fully expanded can be transformed into an equivalent fully expanded version.[1] The Definitions 1 to 3 allow us to characterize the set of *well-formed* PL-CCS programs, which will be the basis for the rest of this paper.

Definition 4 (well-formed PL-CCS program). *A PL-CCS program is* well-formed*, if it is complete, finitely configurable, and fully expanded.*

The rational for the syntactical restrictions leading to Definition 4 is that in a well-formed PL-CCS program we can easily label each variants operator with a unique natural number by parsing over the PL-CCS program and attaching a fresh number to every occurrence of a variants operator. This allows us to precisely define the concept of a variation point: We call a uniquely labeled variants operator with number $i \in \mathbb{N}$, denoted by $\oplus_i$, a *variation point*.

In practical applications, not all combinatorially possible configurations are meaningful or allowed for various non-functional reasons. For example, recall the example from Section 2.1: An OEM might always provide the more advanced keyless locking system whenever a premium infotainment system is selected. Thus, whenever the (optional) infotainment system is chosen, we have to select the keyless variant as well. Such non-functional dependencies between different subsystems are usually captured in a feature model. In their mathematical essence, feature models define a restricted set of configurations. The framework described in this paper does not include such a dependency model and has to be extended to cope with such restrictions. However, to keep the presentation simple, we defer the formal treatment of such feature dependencies to our future work.

2.3 Semantics of a PL-CCS Program

In the following, we define the semantics of a PL-CCS program. More precisely, we introduce three different semantics, the *flat semantics,* the *unfolded semantics,* and the *configured-transitions semantics,* and show how they are related. Basically, the first two semantics are only introduced to motivate and justify the final semantics, the configured-transitions semantics, which will be an appropriate basis for model checking described in Section 3.

Flat Semantics. The *flat semantics* reflects the intuitive understanding of a PL-CCS program: Every PL-CCS program can be understood as the set of all (plain) CCS programs that can be derived by a total configuration of the PL-CCS program. More

[1] This sentence is not to be understood in a mathematical sense, as no semantics for non-fully expanded programs has and will be provided, which does not allow to define *equivalence* precisely.

precisely, given a well-formed PL-CCS program, we choose for every variants operator either the process term on its left- or right-hand side and remove all the unselected terms together with the respective $\oplus$ symbols from the PL-CCS program. For every such configuration, this procedure results in a plain CCS program, which can be understood in the usual way, e.g. with the semantics described by Milner [Mil80].

Technically, given a well-formed PL-CCS program $Prog$ with $n \in \mathbb{N}$ variants operators, we label every variants operator $\oplus$ uniquely with a number in $\{1, \ldots, n\}$. The individual configuration selection made for the i^{th} $\oplus$-operator is stored in the i^{th} entry θ_i of the *configuration vector* $\theta \in \{R, L, ?\}^n$: The entry R represents the selection of the right process, L represents the selection of the left process, and $?$ represents the situation that none of the two alternatives has been selected so far. We call a configuration vector $\theta \in \{R, L, ?\}^n$ *fully configured* if $\forall i \in \{1, \ldots, n\} : \theta_i \neq\ ?$.

Given a well-formed PL-CCS program $Prog$ with n variants operators and a fully configured configuration vector $\theta \in \{R, L, ?\}^n$ we define a function

$$\mathit{config} : \mathcal{P} \times \{R, L, ?\}^n \to \mathcal{R}$$

where $\mathcal{R}$ is the set of CCS programs. The function config reduces $Prog$ to a CCS program V, where V is constructed by removing all terms in $Prog$ which are not selected according to θ.

This allows us to define the flat semantics of a PL-CCS program $Prog$ as

$$[\![Prog]\!]_{Flat} = \left\{ [\![V]\!]_{\text{CCS}} \;\mid\; \exists \theta : (\mathit{config}(Prog, \theta) = V) \right\} \tag{2}$$

where $[\![V]\!]_{CCS}$ denotes the conventional CCS semantics of the CCS program V as defined, e.g., in [Mil80] by means of SOS rules. Due to space limitations, we omit to present the original CCS-SOS rules but refer to the PL-CCS-SOS-rules given in Figure 2, which are of the same form as the original ones but additionally carry a configuration vector.

Note that feature constraints can be incorporated in the flat semantics by considering only appropriate configurations θ in Equation 2.

Unfolded Semantics. Recall that in the flat semantics, a PL-CCS program gives rise to a *set* of transition systems, one for each fully configured configuration. In the unfolded semantics, the meaning of a PL-CCS program is defined by a *single* labeled transition system modeling the behavior of an entire product family. In particular, by combining the behavior of all derivable systems within *one* labeled transition system, it provides the basis for reducing effort in model checking, by considering commonalities between systems. Before defining the unfolded semantics we introduce a suitable transition system:

A *product-line transition system* (PL-LTS) with n variants operators is a tuple $(\mathcal{S}, \mathcal{A}, \Delta, \sigma)$, where $\mathcal{S}$ is a (countably, possibly infinite) set of states, $\mathcal{A}$ is a set of communication actions, and Δ is a finite set of transition relations of the form $\xrightarrow{\alpha,\ \nu} \subseteq \mathcal{S} \times \mathcal{S}$, where $\alpha \in \mathcal{A}$, $\nu \in \{R, L, ?\}^n$, and $\sigma \in \mathcal{S}$ is the start state.

Thus, in a PL-LTS a transition from one state to another is labeled by an action α and an additional (partial) configuration vector ν. However, a transition $s \xrightarrow{\alpha,\ \nu} s'$ represents the set of all transitions $s \xrightarrow{\alpha,\ \nu'} s'$ with ν more general than ν':

$$\frac{P, \nu \xrightarrow{\alpha,\ \nu} P', \nu}{C, \nu \xrightarrow{\alpha,\ \nu} P', \nu}\ ,\ C \stackrel{def}{=} P \qquad (constant\ definition) \tag{3}$$

$$\frac{}{\alpha.P, \nu \xrightarrow{\alpha,\ \nu} P, \nu}\ ,\ \text{for arbitrary } \nu \in \{R, L, ?\}^n \qquad (prefix) \tag{4}$$

$$\frac{P, \nu \xrightarrow{\alpha,\ \nu} P', \nu}{P + Q, \nu \xrightarrow{\alpha,\ \nu} P', \nu} \qquad \textit{(non-deterministic choice (1))} \tag{5}$$

$$\frac{Q, \nu \xrightarrow{\alpha,\ \nu} Q', \nu}{P + Q, \nu \xrightarrow{\alpha,\ \nu} Q', \nu} \qquad \textit{(non-deterministic choice (2))} \tag{6}$$

$$\frac{P, \nu \xrightarrow{\alpha,\ \nu} P', \nu}{(P \parallel Q), \nu \xrightarrow{\alpha,\ \nu} (P' \parallel Q), \nu} \qquad \textit{(parallel composition (1))} \tag{7}$$

$$\frac{Q, \nu \xrightarrow{\alpha,\ \nu} Q', \nu}{(P \parallel Q), \nu \xrightarrow{\alpha,\ \nu} (P \parallel Q'), \nu} \qquad \textit{(parallel composition (2))} \tag{8}$$

$$\frac{P, \nu \xrightarrow{\alpha,\ \nu} P', \nu \qquad Q, \nu \xrightarrow{\bar{\alpha},\ \nu} Q', \nu}{(P \parallel Q), \nu \xrightarrow{\tau,\ \nu} (P' \parallel Q'), \nu} \qquad \textit{(parallel composition (3))} \tag{9}$$

$$\frac{P, \nu \xrightarrow{\alpha,\ \nu} P', \nu}{P[f], \nu \xrightarrow{f(\alpha),\ \nu} P'[f], \nu} \qquad \textit{(re-labeling)} \tag{10}$$

$$\frac{P, \nu \xrightarrow{\alpha,\ \nu} P', \nu}{(P \setminus L), \nu \xrightarrow{\alpha,\ \nu} (P' \setminus L), \nu}\ ,\ \alpha, \bar{\alpha} \notin L \qquad \textit{(restriction)} \tag{11}$$

Fig. 2. SOS rules for unfolded semantics, except of $\oplus$-operator

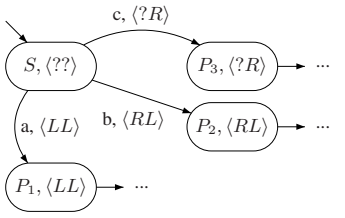
Given two vectors $\nu, \nu' \in \{R, L, ?\}^n$, we call ν *more general* than ν', denoted by $\nu \sqsubseteq \nu'$, if $\forall i \in \{1, \ldots, n\} : \big((\nu_i = ?) \vee (\nu_i = \nu'_i)\big)$. We say that ν *characterizes* the set of configuration vectors $\{\nu' \mid \nu \sqsubseteq \nu'\}$.

Let us now elaborate on the *unfolded semantics* of PL-CCS programs. Similar as for CCS, we define the labeled transition relation by means of enriched SOS rules. The states of the transition system are pairs of PL-CCS process expressions paired with a vector characterizing the configurations under which this state was reached. In order to keep track of the choices for the variants operators the original SOS rules are enriched with a vector ν characterizing the configuration vectors for every transition.

Except for the variants operator $\oplus$, the (original) CCS rules do not influence the construction of the vectors attached to the transitions and are therefore only adjusted in order to be capable of dealing with vectors. The respective rules are given in Figure 2.

For example, rules (3) and (4) express that the execution of an action—specified either directly by action-prefixing as in (4) or indirectly by a constant definition as in rule (3)—can be performed without affecting the current configuration ν, i.e. any state $\alpha.P, \nu$ affords a transition labeled with the action α in every possible configuration ν.

Essential for the unfolded semantics is the treatment of the variants operator $\oplus$: Recall that it is a binary operator which allows to model a selection between two alternative processes where only one will be existing in the final system. Though looking similar to the ordinary CCS $+$-operator (which in a way also models a choice between



(a) PL-LTS

$$\dfrac{\dfrac{\dfrac{}{a.P_1, \langle LL\rangle \xrightarrow{a,\ \langle LL\rangle} P_1, \langle LL\rangle}}{a.P_1 \oplus_1 b.P_2, \langle ?L\rangle \xrightarrow{a,\ \langle LL\rangle} P_1, \langle LL\rangle}}{(a.P_1 \oplus_1 b.P_2) \oplus_2 c.P_3, \langle ??\rangle \xrightarrow{a,\ \langle LL\rangle} P_1, \langle LL\rangle}$$

(b) deduction

Fig. 3. PL-LTS for $S \stackrel{def}{=} (a.P_1 \oplus_1 b.P_2) \oplus_2 c.P_3$ and the deduction of transition $\xrightarrow{a,\langle LL\rangle}$

alternatives), it has to be treated different, for two reasons: First, when a selection has been made, the same selection has to be taken when recursively revisiting the same $\oplus$-operator. Second, the choice has to be "made visibly" in the transition relation, to allow further reasoning on each configuration by model checking.

These two issues are captured by the following two SOS rules for the $\oplus$-operator:

$$\frac{P, \nu|_{i/L} \xrightarrow{\alpha,\ \nu'|_{i/L}} P', \nu'|_{i/L}}{P \oplus_i Q, \nu \xrightarrow{\alpha,\ \nu'|_{i/L}} P', \nu'|_{i/L}}\ ,\ \nu_i \neq R \qquad \textit{(configuration selection (1))} \tag{12}$$

$$\frac{Q, \nu|_{i/R} \xrightarrow{\alpha,\ \nu'|_{i/R}} Q', \nu'|_{i/R}}{P \oplus_i Q, \nu \xrightarrow{\alpha,\ \nu'|_{i/R}} Q', \nu'|_{i/R}}\ ,\ \nu_i \neq L \qquad \textit{(configuration selection (2))} \tag{13}$$

Here, $\nu|_{i/x}$ represents the updated vector ν where the entry at the i^{th} position is replaced by the value $x \in \{R, L\}$. All other entries keep their values, i.e. $\forall\ j \neq i :\ (\nu|_{i/x})_j = \nu_j$. Recall that ν_i yields the i^{th} element of the vector ν. Further note that the respective conditions of the alternative rules prevent the user from selecting a different alternative when re-entering the selection decision due to a recursive process present in CCS.

We define the *unfolded semantics* of a PL-CCS program *Prog*, denoted by $[\![Prog]\!]_{UF}$, as the PL-LTS obtained by applying the SOS rules to the main process identifier.

As an example, Figure 3(a) shows the PL-LTS when applying the *configuration selection* rules to the PL-CCS program starting with the main process definition $S \stackrel{def}{=} (a.P_1 \oplus_1 b.P_2) \oplus_2 c.P_3$. Since the presence of $c.P_3$ in the final configuration only requires to select the right variant at the variation point $\oplus_2$, the corresponding transition to state $P_3, \langle ?R\rangle$ only fixes the second entry of the configuration vector to the value R while leaving any choice for the first entry (?). In contrast to that, the selection of either P_1 or P_2 requires to take two configuration decisions, reflected by the vectors $\langle LL\rangle$ and $\langle RL\rangle$ in the respective states $P_1, \langle LL\rangle$ and $P_2, \langle RL\rangle$. A corresponding deduction

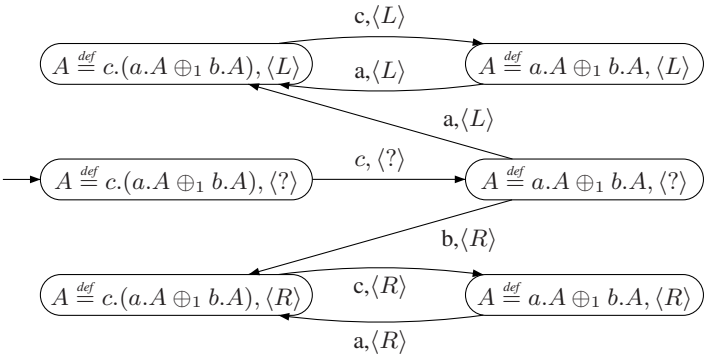


Fig. 4. PL-LTS for the PL-CCS term $A \stackrel{def}{=} c.(a.A \oplus b.A)$

(applying twice Rule 12) for the selection of the variant P_1 is given in Figure 3(b). Note that the derivation shows that the semantics can require several configuration selections for deriving a single transition.

Figure 4 shows an example for the configuration selection rules for recursive process definitions. More specifically, the PL-LTS for the PL-CCS program $A \stackrel{def}{=} c.(a.A \oplus b.A)$ is shown. The state labels correspond to the process term together with the configuration under which they were reached. If the semantics would only depend on the current state's CCS-term (and not additionally on the configuration selected so far), the states at the left and the right column could not be told apart, since the process term is the same for all three states in one column. But since the unfolded semantics keeps track of which configuration was chosen so far, identical PL-CCS terms yield different states in the PL-LTS under different configurations. More precisely, this means that in the state labeled with $A \stackrel{def}{=} a.A \oplus_1 b.A, \langle L \rangle$ the semantics does not allow to have an outgoing transition labeled with $\xrightarrow{b,\langle R \rangle}$, since the dual configuration $\langle L \rangle$ has already been selected.

While the unfolded semantics is easily understood and does indeed represent the behavior of a PL-CCS program within a single transition system, the previous example leads one to suspect that the unfolded semantics yields non-compact transition systems. In the next section, we introduce a configured-transitions semantics, which is based on the unfolded semantics yet yields smaller transition systems.

Let us elaborate on the correctness of the unfolded semantics in a sense made precise below. Therefore, recall that two transition systems are called bisimilar, denoted by $\approx$, when, starting at the initial states, every transition of one system can be simulated by one of the other system and vice versa (see [Mil95] for a precise definition). From a PL-LTS, we obtain for a given configuration vector θ a labeled transition system by projecting to those states and transitions whose vector ν is more general, i.e. where $\nu \sqsubseteq \theta$, and discarding all other transitions. For a PL-CCS program with unfolded semantics $[\![Prog]\!]_{UF}$, let the transition system obtained in this way be denoted by $\Pi_\theta([\![Prog]\!]_{UF})$.

The following theorem states that (modulo bisimulation) the systems given in terms of the flat semantics and the unfolded semantics coincide.

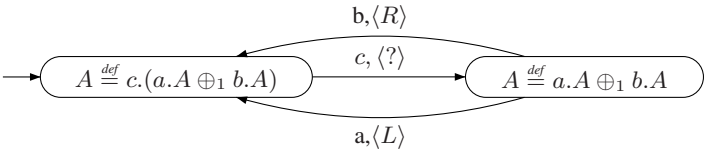


Fig. 5. Configured-transitions semantics for $A \stackrel{def}{=} c.(a.A \oplus b.A)$

Theorem 1 (Correctness of unfolded semantics). *Given a PL-CCS program Prog and a configuration vector θ,*

$$[\![config(Prog, \theta)]\!]_{CCS} \approx \Pi_\theta([\![Prog]\!]_{UF})$$

Due to space limitations we omit the proof here and refer to an extended version of the paper [GLS08].

Configured-transitions Semantics. In the following, we give a further semantics for a PL-CCS program which yields a smaller transition system and, at the same time, caters for model checking the entire product line as described in the next section. The idea is to identify states that have the same PL-CCS process term but only differ in the corresponding configuration vector.

Let $\stackrel{\alpha,\nu}{\Longrightarrow} \subseteq \mathcal{P} \times \mathcal{P}$ be defined by

$$P \stackrel{\alpha,\nu}{\Longrightarrow} P' \text{ iff there exists } \nu' \text{ with } P, \nu' \stackrel{\alpha,\ \nu}{\longrightarrow} P', \nu$$

where $\alpha \in \mathcal{A}$ and $\nu, \nu' \in \{L, R, ?\}^n$ and $\stackrel{\alpha,\ \nu}{\longrightarrow}$ is the relation defined in the previous section.

We define the *Configured-transitions semantics* of a PL-CCS program $Prog$, denoted by $[\![Prog]\!]_{CT}$, as the PL-LTS consisting of states reachable from the main process identifier wrt. $\stackrel{\alpha,\nu}{\Longrightarrow}$ and corresponding transition relations.

Figure 5 shows the transition system for the program $A \stackrel{def}{=} c.(a.A \oplus b.A)$. A comparison with Figure 4 showing the unfolded semantics for the same program shows that the configured-transitions semantics yields indeed smaller transition systems.

For any PL-CCS program $Prog$, every path in $[\![Prog]\!]_{UF}$ corresponds to one execution of one product of the family. This does no longer hold for the paths of $[\![Prog]\!]_{CT}$. For example, the path $cacb$ in the system shown in Figure 5 does not exist in any of the transition systems of $[\![A \stackrel{def}{=} c.(a.A \oplus b.A)]\!]_{Flat}$. However, the interesting property of the configured-transitions semantics is that for every configuration vector θ, the projection of $\Pi_\theta([\![Prog]\!]_{CT})$, similarly defined as for $[\![Prog]\!]_{UF}$, yields the same transition system (modulo isomorphism) as the one obtained when projecting $Prog$ wrt. θ and taking the CCS semantics:

Theorem 2 (Correctness of configured-transitions semantics). *Given a PL-CCS program Prog and a configuration vector θ,*

$$[\![config(Prog, \theta)]\!]_{CCS} = \Pi_\theta([\![Prog]\!]_{CT})$$

A corresponding proof can be found in [GLS08].

3 Model Checking Product Lines

In this section, we introduce a *multi-valued modal version* of the μ-calculus suitable for specifying properties of individual configurations of a PL-CCS program. Furthermore, we sketch a game-based and therefore on-the-fly model checking approach for PL-CCS programs with respect to μ-calculus specifications.

We have chosen to develop our verification approach for specifications in the μ-calculus as it subsumes lineartime temporal logic as well as computation-tree logic as first shown in [EL86, Wol] and nicely summarized in [Dam94]. Therefore we can use our approach also in combination with these logics, and in particular have support for the language SALT [BLS06] used with our industrial partners.

Multi-valued modal μ-calculus combines Kozen's modal μ-calculus [Koz83] and multi-valued μ-calculus as defined by Grumberg and Shoham [SG05] in a way suitable for specifying and checking properties of PL-CCS programs. More specifically, we extend the work of [SG05], which only supports unlabeled diamond and box operators, by providing also action-labeled versions of these operators, which is essential to formulate properties of PL-CCS programs.[2]

A *lattice* is a partially ordered set $(\mathcal{L}, \sqsubseteq)$ where for each $x, y \in \mathcal{L}$, there exists (i) a unique *greatest lower bound* (glb), which is called the *meet* of x and y, and is denoted by $x \sqcap y$, and (ii) a unique *least upper bound* (lub), which is called the *join* of x and y, and is denoted by $x \sqcup y$. The definitions of glb and lub extend to finite sets of elements $A \subseteq \mathcal{L}$ as expected, which are then denoted by $\bigsqcap A$ and $\bigsqcup A$, respectively. A lattice is called *finite* iff $\mathcal{L}$ is finite. Every finite lattice has a least element, called *bottom*, denoted by $\bot$, and a greatest element, called *top*, denoted by $\top$. A lattice is *distributive*, iff $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$, and, dually, $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$. In a *DeMorgan* lattice, every element x has a unique *dual* element $\neg x$, such that $\neg\neg x = x$ and $x \sqsubseteq y$ implies $\neg x \sqsubseteq y$. A complete distributive lattice is called *Boolean* iff the $x \sqcup \neg x = \top$ and $x \sqcap \neg x = \bot$.

While the developments to come do not require to have a Boolean lattice, we will apply them only to the Boolean lattices given by the powerset of possible configurations. In other words, given a set of possible configurations N, the lattice considered is $(2^N, \subseteq)$ where meet, join, and dual of elements, are given by intersection, union, and complement of sets, respectively.

Multi-valued modal μ-calculus. Let $\mathcal{P}$ be a set of *propositional constants*, and $\mathcal{A}$ be a set of *action names*.[3] A *multi-valued modal Kripke structure* (MMKS) is a tuple $\mathcal{T} = (\mathcal{S}, \{\mathcal{R}_\alpha(\,.\,,\,.\,) \mid \alpha \in \mathcal{A}\}, L)$ where $\mathcal{S}$ is a set of states, and $\mathcal{R}_\alpha(\,.\,,\,.\,) : \mathcal{S} \times \mathcal{S} \to \mathcal{L}$ for each $\alpha \in \mathcal{A}$ is a valuation function for each pair of states and action $\alpha \in \mathcal{A}$. Furthermore, $L : \mathcal{S} \to \mathcal{L}^{\mathcal{P}}$ is a function yielding for every state a function from $\mathcal{P}$ to $\mathcal{L}$, yielding a value for each state and proposition. For PL-CCS programs, the idea is that $\mathcal{R}_\alpha(s, s')$ denotes the set of configurations in which there is an α-transition from state

[2] Thus, strictly speaking, we define a multi-valued and multi-modal version of the μ-calculus. However, we stick to a shorter name for simplicity.

[3] So far, for PL-CCS programs, we do not need support for propositional constants. As adding propositions only intricates the developments to come slightly, we show the more general account in the following.

s to s'. It is a simple matter to translate (on-the-fly) the transition system obtained via the configured-transitions semantics into a MMKS.

A Kripke structure in the usual sense can be regarded as a MMKS with values over the two element lattice consisting of a bottom $\bot$ and a top $\top$ element, ordered in the expected manner. Value $\top$ then means that the property holds in the considered state while $\bot$ means that it does not hold. Similarly, $\mathcal{R}_\alpha(s, s') = \top$ reads as there is a corresponding a transition while $\mathcal{R}_\alpha(s, s') = \bot$ means there is no α-transition.

Let $\mathcal{V}$ be a set of propositional variables. Formulae of the *multi-valued modal μ-calculus* in *positive normal form* are given by

$$\varphi ::= \mathit{true} \mid \mathit{false} \mid q \mid \neg q \mid Z \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \langle\alpha\rangle\varphi \mid [\alpha]\varphi \mid \mu Z.\varphi \mid \nu Z.\varphi$$

where $q \in \mathcal{P}$, $\alpha \in \mathcal{A}$, and $Z \in \mathcal{V}$. Let $mv\text{-}\mathfrak{L}_\mu$ denote the set of *closed* formulae generated by the above grammar, where the fixpoint quantifiers μ and ν are variable binders. We will also write η for either μ or ν. Furthermore we assume that formulae are well-named, i.e. no variable is bound more than once in any formula. Thus, every variable Z *identifies* a unique subformula $\mathit{fp}(Z) = \eta Z.\psi$ of φ, where the set $Sub(\varphi)$ of *subformulae* of φ is defined in the usual way.

The semantics of a $mv\text{-}\mathfrak{L}_\mu$ formula is an element of $\mathcal{L}^{\mathcal{S}}$—the functions from $\mathcal{S}$ to $\mathcal{L}$, yielding for the formula at hand and a given state the *satisfaction value*. In our setting, this is the set of configurations for which the formula holds in the given state.

Then the *semantics* $[\![\varphi]\!]^{\mathcal{T}}_{\rho}$ of a $mv\text{-}\mathfrak{L}_\mu$ formula φ with respect to a MMKS $\mathcal{T} = (\mathcal{S}, \{\mathcal{R}_\alpha(\,.\,,\,.\,) \mid \alpha \in \mathcal{A}\}, L)$ and an *environment* $\rho : \mathcal{V} \to \mathcal{L}^{\mathcal{S}}$, which explains the meaning of free variables in φ, is an element of $\mathcal{L}^{\mathcal{S}}$ and is defined as shown in Figure 6. We assume $\mathcal{T}$ to be fixed and do not mention it explicitly anymore. With $\rho[Z \mapsto f]$ we denote the environment that maps Z to f and agrees with ρ on all other arguments. Later, when only closed formulae are considered, we will also drop the environment from the semantic brackets.

The semantics is defined in a standard manner. The only operators deserving a discussion are the $\Diamond$ and $\Box$-operators. Intuitively, $\langle\alpha\rangle\varphi$ is classically supposed to hold in states that have an α-successor satisfying φ. In a multi-valued version, we first consider the value of α-transitions and reduce it (meet it) with the value of φ in the successor state. As there might be different α-transitions to different successor states, we take the best value. For PL-CCS programs, this meets exactly our intuition: A configuration in state s satisfies a formula $\langle\alpha\rangle\varphi$ if it has an α-successor satisfying φ. Dually, $[\alpha]\varphi$ is classically supposed to hold in states for which all α-successors satisfy φ. In a multi-valued version, we first consider the value of α-transitions and increase it (join it) with the value of φ in the successor state. As there might be several different α-successor states, we take the worst value. Again, this meets our intuition for PL-CCS programs: A configuration in state s satisfies a formula $[\alpha]\varphi$ if all α-successors satisfy φ.

The functionals $\lambda f.[\![\varphi]\!]_{\rho[Z\mapsto f]} : \mathcal{L}^{\mathcal{S}} \to \mathcal{L}^{\mathcal{S}}$ are monotone wrt. $\sqsubseteq$ for any Z, φ and $\mathcal{S}$. According to [Tar55], least and greatest fixpoints of these functionals exist.

Approximants of $mv\text{-}\mathfrak{L}_\mu$ formulae are defined in the usual way: if $\mathit{fp}(Z) = \mu Z.\varphi$ then $Z^0 := \lambda s.\bot$, $Z^{\alpha+1} := [\![\varphi]\!]_{\rho[Z\mapsto Z^\alpha]}$ for any ordinal α and any environment ρ, and $Z^\lambda := \bigsqcap_{\alpha<\lambda} Z^\alpha$ for a limit ordinal λ. Dually, if $\mathit{fp}(Z) = \nu Z.\varphi$ then $Z^0 := \lambda s.\top$, $Z^{\alpha+1} := [\![\varphi]\!]_{\rho[Z\mapsto Z^\alpha]}$, and $Z^\lambda := \bigsqcup_{\alpha<\lambda} Z^\alpha$.

$$
\begin{array}{rcl}
[\![true]\!]_\rho & := & \lambda s.\top \\
[\![false]\!]_\rho & := & \lambda s.\bot \\
[\![q]\!]_\rho & := & \lambda s.L(s)(q) \\
[\![\neg q]\!]_\rho & := & \lambda s.\overline{L(s)(q)} \\
[\![Z]\!]_\rho & := & \rho(Z)
\end{array}
\qquad
\begin{array}{l}
[\![\varphi \vee \psi]\!]_\rho := [\![\varphi]\!]_\rho \sqcup [\![\psi]\!]_\rho \\
[\![\varphi \wedge \psi]\!]_\rho := [\![\varphi]\!]_\rho \sqcap [\![\psi]\!]_\rho \\
[\![\langle\alpha\rangle\varphi]\!]_\rho := \lambda s. \bigsqcup\{\mathcal{R}_\alpha(s,s') \sqcap [\![\varphi]\!]_\rho(s')\} \\
[\![[\alpha]\varphi]\!]_\rho := \lambda s. \bigsqcap\{\neg\mathcal{R}_\alpha(s,s') \sqcup [\![\varphi]\!]_\rho(s')\} \\
[\![\mu Z.\varphi]\!]_\rho := \bigsqcap\{f \mid [\![\varphi]\!]_{\rho[Z\mapsto f]} \sqsubseteq f\} \\
[\![\nu Z.\varphi]\!]_\rho := \bigsqcup\{f \mid f \sqsubseteq [\![\varphi]\!]_{\rho[Z\mapsto f]}\}
\end{array}
$$

Fig. 6. Semantics of mv-$\mathfrak{L}_\mu$ formulae

Theorem 3 (Computation of Fixpoints, [Tar55]). *For all MMKS $\mathcal{T}$ with state set $\mathcal{S}$ there is an $\alpha \in \mathbb{O}\mathrm{rd}$ s.t. for all $s \in \mathcal{S}$ we have: if $[\![\eta Z.\varphi]\!]_\rho(s) = x$ then $Z^\alpha(s) = x$.*

The following theorem states that the multi-valued modal semantics of the μ-calculus is indeed suitable for checking the different configurations of a PL-CCS program.

Theorem 4 (Correctness of Model Checking). *For all PL-CCS programs $Prog = (\mathcal{E}, P_1)$, every configuration vector ν, and formulae $\varphi \in mv$-$\mathfrak{L}_\mu$, we have*

$$[\![config(Prog,\nu)]\!]_{CCS} \models \varphi \text{ iff } \nu \in ([\![Prog]\!]_{CT} \models \varphi)(P_1)$$

The proof follows by structural induction on the formula.

While Theorem 3 also implies a way for computing the satisfaction value of an mv-$\mathfrak{L}_\mu$-formula and a given MMKS, this naive fixpoint computation is typically expensive. Game-based approaches originating from the work by [EJS93] and [Sti95] allow model checking in a so-called *on-the-fly* or *local* fashion. In context of multi-valued μ-calculus, the game-based setting becomes technically more involved, as described in detail in [SG05]. Nevertheless, the essence of the game-based approach of computing a satisfaction value based on the so-called *game graph* is similar. For the multi-valued modal μ-calculus, a slight adaption of the approach taken in [SG05] yields game-based approach for the full multi-valued modal μ-calculus.

Due to space limitations, we skip details of the game-based model checking approach for the multi-valued modal μ-calculus.

4 Specification and Verification of a Sample Product-Line

Let us now demonstrate our approach on a simplified version of an industrial case study we have been working on. We consider a product line whose configurations realize different versions of a windscreen wiper system.

Specification. At first, we specify the family of systems, using the formalism introduced in Section 2. The windscreen wiper systems that we specify in our family $WipFam$ are each built of two subcomponents: a rain sensor, $Sensor$, and a windscreen wiper, $Wiper$. Both subcomponents can be realized by two variants, a high and a low one, respectively:

$$WipFam \stackrel{def}{=} Sensor \parallel Wiper \tag{E1}$$

$$Sensor \stackrel{def}{=} SensL \oplus_1 SensH \tag{E2}$$

$$Wiper \stackrel{def}{=} WipL \oplus_2 WipH \tag{E3}$$

The low variant $SensL$ of the sensor is specified as follows:

$$SensL \stackrel{def}{=} non.SensL + ltl.Raining + hvy.Raining + \overline{noRain}.SensL \tag{E4}$$

$$Raining \stackrel{def}{=} non.SensL + ltl.Raining + hvy.Raining + \overline{rain}.Raining \tag{E5}$$

The low variant $SensL$ only detects two different environmental conditions—dry and raining—even though the environment can stimulate the sensor with three different conditions: hvy for heavy rain, ltl for little rain and non for no rain. However, this sensor cannot differ between heavy and little rain, i. e. for this sensor, hvy and ltl have the same effect, as the sensor reaches a process $Raining$ and provides an action $\overline{rain}$, indicating solely the fact that it is raining (without precisely characterizing the intensity). As long as no rain has been detected, the sensor provides the action $\overline{noRain}$, respectively.

The high version of the sensor can distinguish between different degrees of rain intensity, i. e. $SensH$ additionally differentiates heavy rain from little rain. Its PL-CCS specification is given in the following:

$$SensH \stackrel{def}{=} non.SensH + ltl.Medium + hvy.Heavy + \overline{noRain}.SensH \tag{E6}$$

$$Medium \stackrel{def}{=} non.SensH + ltl.Medium + hvy.Heavy + \overline{rain}.Medium \tag{E7}$$

$$Heavy \stackrel{def}{=} non.SensH + ltl.Medium + hvy.Heavy + \overline{hvyRain}.Heavy \tag{E8}$$

In this product line, the sensors can be arbitrarily combined with two variants of windscreen wipers, $WipL$ and $WipH$. In particular, for this example we have no additional non-functional dependencies between the possible variants which would restrict the set of combinatorially possible configurations.

The low version $WipL$ offers two operation modes: (i) a manual mode with perpetual wiper arm movement (action $permWip$), which has to be activated explicitly by the driver, (ii) and a semi-automatic interval mode in which the wiper arm moves at a lower frequency triggered by the rain sensor (via the action $rain$).

$$WipL \stackrel{def}{=} off.WipL + manualOn.Permanent + intvOn.Interval \tag{E9}$$

$$Interval \stackrel{def}{=} noRain.Interval + intvOff.WipL + intvOn.Interval + rain.Wiping + hvyRain.Wiping \tag{E10}$$

$$Wiping \stackrel{def}{=} \overline{slowWip}.Interval + intvOn.Interval \tag{E11}$$

$$Permanent \stackrel{def}{=} \overline{permWip}.Permanent + off.WipL + intvOn.Interval \tag{E12}$$

The high variant $WipH$ can operate at two speeds: slow (action: $\overline{slowWip}$) and fast (action: $\overline{fastWip}$). Here, the wiper arm movement is fully controlled by the rain sensor and adjusts its frequency automatically to the current rain intensity.

$$WipH \stackrel{def}{=} off.WipH + intvOn.AutoIntv \tag{E13}$$

$$AutoIntv \stackrel{def}{=} noRain.AutoIntv + intvOn.AutoIntv + rain.Slow + intvOff.WipH + hvyRain.Fast \tag{E14}$$

$$Slow \stackrel{def}{=} \overline{slowWip}.AutoIntv + intvOn.AutoIntv \tag{E15}$$

$$Fast \stackrel{def}{=} \overline{fastWip}.AutoIntv + intvOn.AutoIntv \tag{E16}$$

The PL-CCS program specifying the entire product line $WipFam$ is given by the equations E1–E16. The whole program $WipFam$ is well-formed, which allows a unique numbering of all (two) variation points as shown by Equations E2 and E3.

Verification. From our example system family $WipFam$, we can derive four different individual systems, as we can combine the subsystem variants arbitrarily. Having specified the family in PL-CCS, we can now apply the model checking approach described in Section 3, in order to verify functional properties for configurations in the system family.

Thinking of a relevant property, for instance, one could possibly be interested in verifying for a windscreen wiping system whether or not a driver is always able to switch to automatic windscreen wiping mode. (Property 1, formalized in Equation 14). Another property could demand the windscreen wiper to wipe fast, once it is raining heavily (Property 2, formalized in Equation 15).

$$\mu X.\langle . \rangle X \vee \langle intvOn \rangle true \tag{14}$$

$$\nu Y.[.]Y \wedge (\neg\langle intvOff \rangle true \vee [hvy]\langle \overline{fastWip} \rangle true) \tag{15}$$

In our example, Property 1 holds for the set of all possible configurations $\langle L, L\rangle$, $\langle R, L\rangle$, $\langle L, R\rangle$,and $\langle R, R\rangle$, which can be denoted by the single vector $\langle ?, ?\rangle$. However, Property 2 is only satisfied in the configuration, in which the high variants of both subsystems are used, i. e. the result of applying the proposed model checking algorithm is the set containing the single configuration vector $\langle R, R\rangle$. Intuitively, it is easy to see why: As the low version of the windscreen wiper does not provide a fast wiping mode, it never provides the output action $\overline{fastWip}$. In consequence, the wind screen wiper can never wipe fast if the low version is used. However, even if the high version of the windscreen wiper is used, but combined with the low version of the rain sensor, the property is still not satisfied. The sensor is not able to provide the output action $\overline{hvyRain}$, which would trigger the wiper to wipe fast. Using our product line specific model checking approach, we are able to identify the configurations which do and do not satisfy a verified property—and which we so far motivated only illustratively—in an automatic way.

5 Conclusion

In this paper, we propose a process algebra approach to software product lines that allows automatic analysis and verification by means of model checking. We introduced PL-CCS as a variant of Milner's CCS designed to model the overall behavior of similar

software systems developed as a software product line. Its semantics can conveniently be defined in terms of multi-valued modal Kripke structures. Furthermore, we introduced multi-valued modal μ-calculus as a property specification language for systems formulated in PL-CCS. Model checking then allows to verify either an entire software product line, or, to point out which variants of the product line do not meet given correctness properties. We are currently working on algebraic properties of PL-CCS, on the integration of a dependency model for modeling feature constraints, as well as on an implementation of the proposed model checking approach.

Acknowledgement. We thank Mila Mejster-Cederbaum for valuable comments on an earlier draft of this paper.

References

[BLS06] Bauer, A., Leucker, M., Streit, J.: SALT—structured assertion language for temporal logic. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260. Springer, Heidelberg (2006)

[CN02] Clements, P., Northrop, L.: Software Product Lines. Practices and Patterns. Addison-Wesley, Reading (2002)

[Dam94] Dam, M.: CTL* and ECTL* as fragments of the modal μ-calculus. Theoretical Computer Science 126(1), 77–96 (1994)

[EJS93] Emerson, E.A., Jutla, C.S., Sistla, A.P.: On model-checking for fragments of mu-calculus. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 385–396. Springer, Heidelberg (1993)

[EL86] Emerson, E.A., Lei, C.L.: Efficient model checking in fragments of the propositional μ-calculus. In: Symposium on Logic in Computer Science (LICS 1986), Washington, D.C., USA, June 1986, pp. 267–278. IEEE Computer Society Press, Los Alamitos (1986)

[FUB07] Fischbein, D., Uchitel, S., Braberman, V.: A foundation for behavioural conformance in software product line architectures. In: Proceedings of the 2nd Workshop on the Role of Software Architecture for Testing and Analysis (2007)

[GLS08] Gruler, A., Leucker, M., Scheidemann, K.: Modelling and Model Checking Software Product Lines. Technical Report TUM-I0806, Technische Universität München (February 2008)

[KNK05] Kishi, T., Noda, N., Katayama, T.: Design verification for product line development. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 150–161. Springer, Heidelberg (2005)

[Koz83] Kozen, D.: Results on the propositional mu-calculus. Theoretical Computer Science 27, 333–354 (1983)

[LKF05] Li, H.C., Krishnamurthi, S., Fisler, K.: Modular verification of open features using three-valued model checking. Automated Software Engineering (2005)

[LNW07] Larsen, K.G., Nyman, U., Wasowski, A.: Modal I/O automata for interface and product line theories. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 64–79. Springer, Heidelberg (2007)

[LT91] Larsen, K.G., Thomsen, B.: Partial specifications and compositional verification. Theor. Comput. Sci. 88(1), 15–32 (1991)

[MC01] Majster-Cederbaum, M.E.: Underspecification for a simple process algebra of recursive processes. Theor. Comput. Sci. 266(1-2), 935–950 (2001)

[Mil80] Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)

[Mil95] Milner, R.: Communication and concurrency. Prentice Hall International (UK) Ltd., Hertfordshire (1995)

[PBvdL05] Pohl, K., Böckle, G., van der Linden, F. (eds.): Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Berlin (2005)

[SG05] Shoham, S., Grumberg, O.: Multi-valued model checking games. In: Peled, D.A., Tsay, Y.-K. (eds.) ATVA 2005. LNCS, vol. 3707, pp. 354–369. Springer, Heidelberg (2005)

[Sti95] Stirling, C.: Local model checking games. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 1–11. Springer, Heidelberg (1995)

[Tar55] Tarski, A.: A lattice-theoretical fixpoint theorem and its application. Pacific J.Math. 5, 285–309 (1955)

[VN98] Veglioni, S., De Nicola, R.: Possible worlds for process algebras. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 179–193. Springer, Heidelberg (1998)

[Wol] Wolper, P.: A translation from full branching time temporal logic to one letter propositional dynamic logic with looping (unpublished manuscript)

Semantic Foundations and Inference of Non-null Annotations

Laurent Hubert[1], Thomas Jensen[1], and David Pichardie[2]

[1] CNRS/IRISA, France
[2] INRIA Rennes - Bretagne Atlantique/IRISA, France

Abstract. This paper proposes a semantics-based automatic null pointer analysis for inferring non-null annotations of fields in object-oriented programs. The analysis is formulated for a minimalistic OO language and is expressed as a constraint-based abstract interpretation of the program which for each field of a class infers whether the field is definitely non-null or possibly null after object initialization. The analysis is proved correct with respect to an operational semantics of the minimalistic OO language. This correctness proof has been machine checked using the Coq proof assistant. We also prove the analysis complete with respect to the non-null type system proposed by Fähndrich and Leino, in the sense that for every typable program the analysis is able to prove the absence of null dereferences without any hand-written annotations. Experiments with a prototype implementation of the analysis show that the inference is feasible for large programs.

1 Introduction

A common source of exceptional program behaviour is the dereferencing of null references (also called null pointers), resulting in segmentation faults in C or null pointer exceptions in Java. Even if such exceptions are caught, the presence of exception handlers creates an additional amount of potential branching which in turn implies that: 1) fewer optimizations are possible and 2) verification is more difficult (bigger certification conditions, implicit flow in information flow verification, etc.). Furthermore, the Java virtual machine is obliged to perform run-time checks for non-nullness of references when executing a number of its bytecode instructions, thereby incurring a performance penalty.[1] For all these reasons, a static program analysis which can guarantee before execution of the program that certain references will definitely be non-null is useful.

Some non-null type systems for the Java language have been proposed [6,15]. Although non-null annotations are not used by the compiler yet, this could likely change in the future, in which case it becomes crucial to prove the soundness of the logic underlying the type checker. Furthermore, although non-null annotations are not yet mandatory, automatic type inference would help to retrofit

[1] Although hardware traps are used for free whenever possible, explicit non-nullness tests are still required as explained in [13].

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 132–149, 2008.

legacy code, to lower the annotation burden on the programmer, to document the code and to verify existing code.

While some object oriented languages ensure all fields are initialized and objects are not read before being fully initialized, this is not the case in Java. More precisely, there are three aspects that complicate non-nullness analysis: 1) fields can be accessed during object construction (before or after their initialization) which means all fields contain null value before their first assignment; 2) no guarantee is given on the initialization of fields at the end of the constructor so if a field has not been initialized it contains a null value; and 3) an object being initialized can be stored in fields or passed as an argument of methods.

The first aspect means a naive flow-insensitive heap abstraction is not sufficient as all fields are null at some stage and hence would be annotated as possibly null. The second aspect makes a special initialization analysis necessary in order to track fields that are certain to be initialized during the construction. Those fields can safely be considered as non-null unless they are explicitly assigned a null value. All other fields might not have been initialized and must be considered as possibly null.

The third aspect was observed in [6] and concerns the problem where a virtual method `A.m()` is being redefined in a sub-class `B` by a method referencing a field `f` that is local to the class `B`. If the constructor of a super-class `A` executes `this.m()` on an object of class `B`, it calls a method that dereferences field `f` before the constructor of `B` has had the possibility of initializing this field. To solve this problem, references to objects under construction need to be tracked, e.g. by a tag indicating their state.

Related Work. Freund and Mitchell have proposed in [9] a *type system* combined with a data-flow analysis to ensure the correct initialization of objects. The goal is to formalize the initialization part of Java bytecode verification. In this respect it is different from our analysis, which is focused on field initialization and on their nullness property. Fähndrich and Leino in [6] have proposed another *type system* also combined with a data-flow analysis to ensure a correct manipulation of references with respect to a nullness property. This system is presented in Sect. 6 where we compare our inference analysis to their type system and give examples of how our analysis infers more precise types than what their system is able to check. More recently, Fähndrich and Xia have proposed another *type system* introducing *delayed* initialization [7]. It generalizes the previous one and allows to prove some properties that our analysis cannot, like initialization of circular data structures, but it has also the same loss of precision discussed in Sect. 6.

Some works are focused on *local type inference*, *i.e.* inferring nullness property for blocks of code from the guards. This is notably the case of FindBugs [11,10] and of the work by Male *et al.* [15]. They rely on *path-sensitive* analysis and the treatment of field initialization is very weak.

To *infer type annotations*, Houdini [8] generates a set of possible annotations (non-null annotations among others) for a given program and uses ESC/Java [14] to refute false assumptions. CANAPA [3] lowers the burden of annotating a

program by propagating some non-null annotations. It also relies on ESC/Java to infer where annotations are needed. Those two annotation assistants have a simplified handling of objects under construction and are intented to be used by the developer to debug and not to certify programs. Indeed, they rely on ESC/Java [14], which is not sound (nor complete). JastAdd [5] is a tool to infer annotations for a simplified version of Fähndrich and Leino's type system.[2]

Finally, another approach to lower the amount of annotations is proposed by Chalin and James [2]. They suggest to consider the references as non-null by default, so the developer has only to explicitly annotate nullable references. Despite the lower amount of annotation needed, about 1/3 of declarations still need annotations which can represent a substantial amount of annotations in legacy code.

Contributions. The non-null reference analysis presented here makes the following contributions.

- The analysis is fully automatic so there is no annotation burden for the programmer.
- The soundness of the analysis is proved formally with respect to an operational semantics. This is the first formal correctness proof for this kind of analyses. Furthermore, this proof has been checked mechanically using the Coq proof assistant.
- We provide a detailed comparison with Fähndrich and Leino's type system, which is a reference among the nullness program analyses. The completeness with respect to their type system is proved. In this way, the correctness proof of our analysis also provides a formal proof of correctness of the type system of Fähndrich and Leino. We also show that our analysis can be slightly more precise than their type system.
- The analysis is modular: a program can be analysed without analyzing the libraries if they have already been annotated.

Outline. Section 2 presents the syntax and semantics of the simple OO language we use to formalize our analysis. Section 3 presents the system of constraints of the analysis and Sect. 4 gives the proof of soundness. Section 5 proves the constraint system has a least fixpoint and discusses the modularity. Section 6 then presents Fähndrich and Leino's type system and proves the completeness of our analysis with respect to their type system. Section 7 gives some details on the adaptation of the analysis to the Java bytecode level and Sect. 8 concludes.

2 Syntax and Semantics

We define a minimalistic language[3] to analyze the flow of references in object-based languages. A program is a collection of classes, arranged in a class hierarchy

[2] The treatment of objects under initialization is simplified and initializations done by methods (called from the constructor) are not taken in account.

[3] The language is closed to the Java bytecode language but without any operand stack. Removing the operand stack avoid to introduce an alias analysis which is needed at the bytecode level, for instance, to know if a stack variable is an alias of **this**.

via a transitive subclass relation $\prec$ (we write $\preceq$ for its reflexive closure). We only consider single inheritance (where $\preceq$ can be embedded in a sup-semilattice structure). The language, as described in Fig. 1 has two kinds of expressions E: variables and references to fields. Assignments are either to variables or to fields. New objects are created using the **new** instruction which takes as arguments the name C of a class, the vector $\boldsymbol{\alpha}$ of the types of the parameters (used to deal with overloading) and a list of expressions. There is a conditional instruction which may non-deterministically branches to a given program label. Finally, the instructions $x.\mathrm{ms}(E,\ldots,E)$ and **return** represent method invocation and return, respectively. Methods invoked are found with a lookup procedure that looks for a method depending on its signature and the class of the current object. A method descriptor ms and a method identifier m are both of the form $\{\!|C, \mathrm{mn}, \boldsymbol{\alpha} \to \beta|\!\}$, where mn is a method name, $\boldsymbol{\alpha} \to \beta$ is a type signature (to simplify the presentation we here restrict β to **void**) and C is, for the method descriptor, the type of the reference on which the method is called and, for the method identifier, the class where the method has been defined. We use name(ms) and name(m) to retrieve mn and class(ms) and class(m) to retrieve C. A method signature is the couple $(\mathrm{mn}, \boldsymbol{\alpha} \to \beta)$. A method signature can correspond to many methods, a method descriptor corresponds to at most one method and a method identifier corresponds to only one method. As a notation abuse, m will be used as the method identified instead of the identifier itself. A method is composed of a method signature, a list of parameters and a method body consisting of a labelled list of instructions. A class is composed of fields and method definitions. For a given field f class(f) gives the class in which f is defined. The set of fields declared in a class C is written fields(C). Each program contains a special method **main** used to start the execution.

The operational semantics of our minimalistic language is defined by the inference rules in Fig. 1 which specify a (small-step) transition relation $\rightarrow_{\mathrm{m}}$ for each method m and a (big-step) evaluation relation $\Downarrow_{h,l}$ for expressions relative to a heap h and local variables l.

$$\Downarrow_{h,l} \subseteq E \times (\mathrm{Val} + \{\Omega\}) \qquad \rightarrow_m \subseteq \mathrm{State} \times (\mathrm{State} + \mathrm{Heap} + \{\Omega\})$$

The semantics uses an explicit error element Ω to signal the dereferencing of a null reference. Alternatively, we could have let the semantics "get stuck" when it encounters an error but our choice of propagating an error element to method boundaries facilitates the correctness proof of the null pointer analysis to follow. For space reasons, we only detail the error case for method invocation; all other instructions may lead to similar error steps. The analysis keeps track of how objects are initialized and, in order to prove its correctness, the semantics instruments the fields of objects with flags **def** or **undef** to track the history of field initializations. A field flagged as **undef** contains necessarily null. This instrumentation is transparent in the sense that it does not affect the behavior of a program and will be used later to prove the correctness of the analysis. In Java bytecode, unlike fields, local variables do not have a default value and the bytecode verifier ensures uninitialized local variables are never read. We formalized this by using $\perp$ as the default value for local variables and by ensuring with

the semantics $\bot$ is never read. We also model object initialization to ensure a constructor of class C terminates only when at least one constructor of each ancestor has been called. To do so, we keep for each object the set of classes whose constructor has been called. The semantics is stuck if a constructor violates that policy. This means we only consider programs where objects are correctly initialized.

Notation: We write $\mathbb{F}$, $\mathbb{V}$, Loc and $\mathbb{N}$ for fields, variables, memory addresses and program points, respectively. We consider that a field includes the class name where it has been defined. We note $h(r)(f)$, $\text{class}(h(r))$ and $\text{history}(h(r))$ the accesses to the first, second and third components of the heap cell $h(r)$, respectively. Function $\text{upd}(h, r, f, (v, \mathbf{def}))$ sets the field f of the object at location r to the value v and marks it as defined. The expression $\text{default}(C)$ denotes a new object of class C where all fields contain null values and are **undef** (the history of such object is empty). The function set2Def_C sets all fields (in $\text{fields}(C)$) of an object to **def**. The function addHistory_C adds a class name C to the initialization history of an object. The star transition $\rightarrow^*_{\text{m}}$ corresponds to the transitive closure of $\rightarrow_{\text{m}}$.

A program is considered *null-pointer error safe* if the execution of the method **main** never reaches an error state, starting from an empty heap with only one (uninitialised) object in it and all local variable (except **this**) with the default value $\bot$.

Definition 1. *A program P is said to be* null-pointer error safe *if for all location r,*

$$\langle 0, \bot[\boldsymbol{this} \mapsto r], h_0\rangle \rightarrow^*_{\boldsymbol{main}} s \text{ implies } s \neq \Omega$$

where h_0 is the heap of domain $\{r\}$ with $h_0(r) = \text{default}(\text{class}(\boldsymbol{main}))$.

Note that since error states are propagated after method calls, this definition of safe program implies that no dereferencing of a null reference will occur during the execution of the program.

3 Null-Pointer Analysis

We will now present the analysis that is able to prove a program is null-pointer error safe. As mentioned in the introduction, although the analysis annotates local variables, method parameters and return value, its purpose is to annotate the fields.

3.1 Abstract Domains

In this section we define an analysis that for each class of a well-typed program infers annotations about nullity of its fields together with pre- and post-conditions for its methods. The basic properties of interest are NotNull — meaning "definitely not-null" — and MayBeNull — meaning possibly a null reference. The field annotations computed by the analysis are represented as a heap abstraction

Syntax

$$
\begin{aligned}
E &::= x \mid E.F \\
I &::= x \leftarrow E \mid x.f \leftarrow E \mid x \leftarrow \textbf{new } (C, \boldsymbol{\alpha})(E, \ldots, E) \mid \textbf{if } (\star) \texttt{ jmp} \mid x.\mathrm{ms}(E, \ldots, E) \mid \textbf{return} \\
M &::= \mathrm{mn}(x_1, \ldots, x_n) \ \{\ I; \ldots; I\ \} \\
C &::= \{|\mathsf{fields} : \{f; \ldots; f\}; \mathsf{methods} : \{M; \ldots; M\}|\} \\
P &::= (C\ \cdots\ C, \prec)
\end{aligned}
$$

Domains

$$
\begin{gathered}
\mathrm{Val} = \mathrm{Loc} + \{\textbf{null}\} \qquad \mathrm{Object} = \mathbb{F} \rightharpoonup \mathrm{Val} \times \{\textbf{def}, \textbf{undef}\} \\
\mathrm{LocalVar} = \mathbb{V} \rightharpoonup \mathrm{Val} + \{\bot\} \qquad \mathrm{Heap} = \mathrm{Loc} \rightharpoonup \mathrm{Object} \times \mathbb{C} \times \wp(\mathbb{C}) \\
\mathrm{State} = (\mathbb{N} \times \mathrm{LocalVar} \times \mathrm{Heap}) + \Omega
\end{gathered}
$$

Expression Evaluation

$$
\frac{l(x) \neq \bot}{x \Downarrow_{h,l} l(x)} \qquad \frac{e \Downarrow_{h,l} r \qquad r \in \mathrm{dom}(h) \qquad f \in \mathrm{dom}(h(r))}{e.f \Downarrow_{h,l} h(r)(f)} \qquad \frac{e \Downarrow_{h,l} v \qquad v \in \{\textbf{null}, \Omega\}}{e.f \Downarrow_{h,l} \Omega}
$$

Operational Semantics: Normal Cases

$$
\frac{\boxed{P_{\mathrm{m}}[i] = x \leftarrow e} \qquad e \Downarrow_{h,l} v \qquad x \neq \textbf{this}}{\langle i, l, h\rangle \rightarrow_{\mathrm{m}} \langle i+1, l[x \mapsto v], h\rangle} \qquad \frac{\boxed{P_{\mathrm{m}}[i] = x.f \leftarrow e} \qquad l(x) \in \mathrm{dom}(h) \qquad f \in \mathrm{dom}(h(l(x))) \qquad e \Downarrow_{h,l} v}{\langle i, l, h\rangle \rightarrow_{\mathrm{m}} \langle i+1, l, \mathrm{upd}(h, l(x), f, (v, \textbf{def}))\rangle}
$$

$$
\frac{\boxed{P_{\mathrm{m}}[i] = x \leftarrow \textbf{new } (C, \boldsymbol{\alpha})(e_1, \ldots, e_n)} \qquad \forall i,\ e_i \Downarrow_{h,l} v_i \qquad r \notin \mathrm{dom}(h) \qquad x \neq \textbf{this} \qquad \langle 0, \bot[\textbf{this} \mapsto r,\ \{i \mapsto v_i\}_{i=1..n}], h[r \mapsto \mathrm{default}(C)]\rangle \rightarrow^{*}_{\{|C, \textbf{init}, \boldsymbol{\alpha} \rightarrow \textbf{void}|\}} h'}{\langle i, l, h\rangle \rightarrow_{\mathrm{m}} \langle i+1, l[x \mapsto r], h'\rangle}
$$

$$
\frac{\boxed{P_{\mathrm{m}}[i] = x.\mathrm{ms}(e_1, \ldots, e_n)} \qquad \forall i,\ e_i \Downarrow_{h,l} v_i \qquad l(x) \in \mathrm{dom}(h) \qquad \mathrm{m}' = \mathrm{lookup}(\mathit{class}(h(l(x))), \mathrm{ms}) \qquad \mathrm{class}(h(l(x))) \preceq \mathrm{class}(\mathrm{ms}) \qquad \langle 0, \bot[\textbf{this} \mapsto l(x),\ \{i \mapsto v_i\}_{i=1..n}], h\rangle \rightarrow^{*}_{\mathrm{m}'} h' \qquad \mathrm{name}(\mathit{ms}) = \textbf{init} \Rightarrow x = \textbf{this}}{\langle i, l, h\rangle \rightarrow_{\mathrm{m}} \langle i+1, l, h'\rangle}
$$

$$
\frac{\boxed{P_{\mathrm{m}}[i] = \textbf{if jmp}}}{\langle i, l, h\rangle \rightarrow_{\mathrm{m}} \langle \mathrm{jmp}, l, h\rangle} \qquad \frac{\boxed{P_{\mathrm{m}}[i] = \textbf{if jmp}}}{\langle i, l, h\rangle \rightarrow_{\mathrm{m}} \langle i+1, l, h\rangle}
$$

$$
\frac{\boxed{P_{\mathrm{m}}[i] = \textbf{return}} \qquad C = \mathrm{class(m)} \qquad \mathrm{name(m)} = \textbf{init} \Rightarrow \forall A, C \prec A \Rightarrow A \in \mathrm{history}(l(\textbf{this})) \qquad \mathrm{name(m)} = \textbf{init} \Rightarrow h' = h[l(\textbf{this}) \mapsto \mathrm{addHistory}_C(\mathrm{set2Def}_C(h(l(\textbf{this}))))] \qquad \mathrm{name(m)} \neq \textbf{init} \Rightarrow h' = h}{\langle i, l, h\rangle \rightarrow_{\mathrm{m}} h'}
$$

Operational Semantics: Error Cases (Selected Rules)

$$
\frac{\boxed{P_{\mathrm{m}}[i] = x.\mathrm{ms}(e_1, \ldots, e_n)} \qquad \forall i,\ e_i \Downarrow_{h,l} v_i \qquad l(x) \in \mathrm{dom}(h) \qquad \mathrm{m}' = \mathrm{lookup}(\mathrm{class}(h(l(x)))), \mathrm{ms}) \qquad \mathrm{class}(h(l(x))) \preceq \mathrm{class}(\mathrm{ms}) \qquad \langle 0, \bot[\textbf{this} \mapsto l(x),\ \{i \mapsto v_i\}_{i=1..n}], h\rangle \rightarrow^{*}_{\mathrm{m}'} \Omega \qquad \mathrm{name(m)} = \textbf{init} \Rightarrow x = \textbf{this}}{\langle i, l, h\rangle \rightarrow_{\mathrm{m}} \Omega}
$$

$$
\frac{\boxed{P_{\mathrm{m}}[i] = x.\mathrm{ms}(e_1, \ldots, e_n)} \qquad l(x) = \textbf{null} \vee \exists i,\ e_i \Downarrow_{h,l} \Omega}{\langle i, l, h\rangle \rightarrow_{\mathrm{m}} \Omega}
$$

Fig. 1. Syntax and semantics of the language

$H^\sharp \in \text{Heap}^\sharp$ which provides an abstraction for all fields of all initialized objects and all initialized fields of objects being initialized.[4]

As explained in the Introduction, object initialization requires a special treatment because all fields are null when the object is created so this would lead a simple-minded analysis to annotate all fields as MayBeNull. However, we want to infer non-null annotations for all fields which are initialized explicitly to be so in a constructor. In other words, fields that are not initialized in a constructor are annotated as MayBeNull and all other fields have a type compatible with the value they have been initialized with.

To this end, the analysis tracks field initializations of the current object in constructors (and methods called from constructors). This is done via an abstraction of the **this** reference by a domain $\text{TVal}^\sharp$ which maps each of the fields declared in the current class to Def or UnDef. To allow strong updates, we need a flow-sensitive abstraction so to each program point we map a such abstraction ($T^\sharp$).

Abstract Domains

$\text{Val}^\sharp = \{\text{Raw}(Y) \mid Y \in \text{Class}\} \cup \{Raw^-, \text{NotNull}, \text{MayBeNull}\}$			
$\text{Def}^\sharp = \{\text{Def}, \text{UnDef}\}$	$\text{TVal}^\sharp = \mathbb{F} \rightharpoonup \text{Def}^\sharp$	$\text{Heap}^\sharp = \mathbb{F} \to \text{Val}^\sharp$	$\text{LocalVar}^\sharp = \mathbb{V} \to \text{Val}^\sharp$
$\text{Method}^\sharp = \mathbb{M} \to \{\lvert \mathsf{this} \in \text{TVal}^\sharp;\ \mathsf{args} \in (\text{Val}^\sharp)^*; \mathsf{post} \in \text{TVal}^\sharp \rvert\}$			
$\text{State}^\sharp = \text{Method}^\sharp \times \text{Heap}^\sharp \times \left(\mathbb{M} \times \mathbb{N} \rightharpoonup \text{TVal}^\sharp\right) \times \left(\mathbb{M} \times \mathbb{N} \rightharpoonup \text{LocalVar}^\sharp\right)$			

Selected Partial Orders

$\text{Val}^\sharp$		$\text{Def}^\sharp$
$\dfrac{x \in \text{Val}^\sharp}{\text{NotNull} \sqsubseteq x}$	$\dfrac{}{\text{Raw}(X) \sqsubseteq Raw^-}$	
$\dfrac{X \preceq Y}{\text{Raw}(X) \sqsubseteq \text{Raw}(Y)}$	$\dfrac{x \in \text{Val}^\sharp}{x \sqsubseteq \text{MayBeNull}}$	$\dfrac{}{\text{Def} \sqsubseteq \text{UnDef}}$

Fig. 2. Abstract domains and selected partial orders

References are then abstracted by a domain $\text{Val}^\sharp$ which incorporate the "raw" references from [6]. A Raw^- value denotes a non-null reference of an object being initialized, which does not yet respect his invariant (*e.g.* Raw^- can be used as a property of `this` when it occurs in constructors). If a reference is known to have all its fields declared in class X and in the parents of X initialized, then the reference is $\text{Raw}(X)$. The inclusion of "raw" references allows the manipulation of objects during initialization because the analysis can use the fact that for an object of type $\text{Raw}(X)$, only fields declared in X and above have a valid annotation in the abstract heap $H^\sharp$. A NotNull value denotes a non-null reference that has finished its initialization.

[4] We consider fields *initialized* when they have been assigned a value, whereas we consider an object *initialized* when it has returned from its constructor.

Figure 2 defines formally each abstract domain. The flow of references through local variables is analysed with a flow-sensitive abstraction of each variable ($L^\sharp \in \text{LocalVar}^\sharp$). The analysis also infers method annotations $M^\sharp \in \text{Method}^\sharp$. For any method m, $M^\sharp(\text{m})[\mathsf{this}]$ is an approximation of the initialization state of **this** before the execution of m, while $M^\sharp(\text{m})[\mathsf{post}]$ gives the corresponding approximation at the end of the execution. $M^\sharp(\text{m})[\mathsf{args}]$ approximates the parameters of the method, taking into account all the context in which m may actually be invoked. We only give the definition of the partial order for $\text{Val}^\sharp$ and $\text{Def}^\sharp$. The other orders are defined in a standard way using the canonical orders on partial functions, products and lists. The final domain $\text{State}^\sharp$ is hence equipped with a straightforward lattice structure.

3.2 Inference Rules

The analysis is specified via a set of inference rules, shown in Fig. 3, which define a judgment

$$M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\text{m}, i) \ : \ \mathit{instr}$$

for when the abstract state $(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$ is coherent with instruction *instr* at program point (m, i). For each such program point, this produces a set of constraints over the abstract domain $\text{State}^\sharp$, whose solutions constitute the correct analysis results. The rules make use of an abstract evaluation function for expressions (explained below) that we write $[\![e]\!]^\sharp$. An example of a program with the corresponding constraints are given in Sect. 3.3.

Assignment to a local variable (rule (1)) simply assigns the abstract value associated with the expression to the local variable in the abstract environment $L^\sharp$. Assignment to a field (2) can either be to a field of the current object, in which case the field becomes "defined", or to another object. In both cases, the abstract heap $H^\sharp$ is augmented with the value of the expression as a possible value for the field. When a return value is encountered (3) in a constructor, all still-undefined fields are explicitly set to MayBeNull. For a `new` instruction (4), a new abstraction $\top_C$ is built for the pre-condition on `this` of the constructor. The first argument (`this`) is set to Raw^-. The result of the constructor is known to be NotNull (the object is fully initialized). The method call (6) uses conditional constraints to distinguish a number of cases, depending on whether the call is to a method in the current class and the receiving object is the current object `this` or not. We use $\text{Raw}(\text{super}(C))$ to denote the Raw type just above $\text{Raw}(C)$.[5]

The whole constraint system of a program P is then formally defined by the judgement $M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P$. We write $\text{overrides}(\text{m}', \text{m})$ when the method m' overrides m. In such case we require a contravariant property between the parameters of m and m'. Any overriding $\text{overrides}(\text{m}', \text{m})$ need also to invalidate (in term of precision) $M^\sharp(\text{m})[\mathsf{post}]$ because a virtual call to m could lead to the execution of m' which is not able (by definition of $T^\sharp$) to track the initialization of fields declared in class(m). A similar constraint is required for $M^\sharp(\text{m}')[\mathsf{this}]$ because for a virtual call to m we have only constrained $M^\sharp(\text{m})[\mathsf{this}]$ and not

[5] If C is the root of the class hierarchy (Object in Java), then $\text{Raw}(\text{super}(C)) = Raw^-$.

Inference Rules

$$\frac{T^\sharp(\mathrm{m},i) \sqsubseteq T^\sharp(\mathrm{m},i+1) \qquad L^\sharp(\mathrm{m},i)[x \mapsto [\![e]\!]^\sharp] \sqsubseteq L^\sharp(\mathrm{m},i+1)}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m},i) \ : \ x \leftarrow e} \tag{1}$$

$$\frac{\begin{array}{c}\text{if } x = \mathbf{this} \wedge f \in \text{fields}(\text{class}(\mathrm{m})) \text{ then } T^\sharp(\mathrm{m},i)[f \mapsto \text{Def}] \text{ else } T^\sharp(\mathrm{m},i) \sqsubseteq T^\sharp(\mathrm{m},i+1) \\ L^\sharp(\mathrm{m},i) \sqsubseteq L^\sharp(\mathrm{m},i+1) \qquad [\![e]\!]^\sharp \sqsubseteq H^\sharp(f)\end{array}}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m},i) \ : \ x.f \leftarrow e} \tag{2}$$

$$\frac{\begin{array}{c}T^\sharp(\mathrm{m},i) \sqsubseteq M^\sharp(\mathrm{m})[\mathsf{post}] \\ \text{name}(\mathrm{m}) = \mathbf{init} \Rightarrow \forall f \in \text{fields}(\text{class}(\mathrm{m})).(T^\sharp(\mathrm{m},i)(f) = \text{UnDef} \Rightarrow \text{MayBeNull} \sqsubseteq H^\sharp(f))\end{array}}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m},i) \ : \ \mathbf{return}} \tag{3}$$

$$\frac{\begin{array}{c}\mathrm{m}' = \{|C, \mathbf{init}, \boldsymbol{\alpha} \to \mathbf{void}|\} \\ \top_C \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{this}] \qquad Raw^- :: [\![e_1]\!]^\sharp :: \cdots :: [\![e_j]\!]^\sharp \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{args}] \\ L^\sharp(\mathrm{m},i)[x \mapsto \text{NotNull}] \sqsubseteq L^\sharp(\mathrm{m},i+1) \qquad T^\sharp(\mathrm{m},i) \sqsubseteq T^\sharp(\mathrm{m},i+1)\end{array}}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m},i) \ : \ x \leftarrow \mathbf{new}\ (C, \boldsymbol{\alpha})(e_1, \ldots, e_j)} \tag{4}$$

$$\frac{\begin{array}{c}L^\sharp(\mathrm{m},i) \sqsubseteq L^\sharp(\mathrm{m},i+1) \qquad L^\sharp(\mathrm{m},i) \sqsubseteq L^\sharp(\mathrm{m},\text{jmp}) \\ T^\sharp(\mathrm{m},i) \sqsubseteq T^\sharp(\mathrm{m},i+1) \qquad T^\sharp(\mathrm{m},i) \sqsubseteq T^\sharp(\mathrm{m},\text{jmp})\end{array}}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m},i) \ : \ \mathbf{if}\ \text{jmp}} \tag{5}$$

$$\frac{\begin{array}{c}\mathrm{m}' = \text{lookup}(\text{class}(\text{ms}), \text{ms}) \qquad C' = \text{class}(\mathrm{m}') \qquad C = \text{class}(\mathrm{m}) \\ \left(\begin{array}{l}\text{if } x = \mathbf{this} \wedge C = C' \wedge [\![x]\!]^\sharp \sqsubseteq \text{Raw}(\text{super}(C')) \wedge M^\sharp(\mathrm{m}')[\mathsf{post}] \sqcap T^\sharp(\mathrm{m},i) \sqsubseteq \bot_{C'} \\ \text{then } L^\sharp(\mathrm{m},i)[x \mapsto \text{Raw}(C') \sqcap [\![x]\!]^\sharp] \sqsubseteq L^\sharp(\mathrm{m},i+1) \\ \text{else if name}(\mathrm{m}') = \mathbf{init} \text{ then } L^\sharp(\mathrm{m},i)[x \mapsto \text{Raw}(C')] \sqsubseteq L^\sharp(\mathrm{m},i+1) \\ \text{else } L^\sharp(\mathrm{m},i) \sqsubseteq L^\sharp(\mathrm{m},i+1)\end{array}\right) \\ \left(\begin{array}{l}\text{if } x = \mathbf{this} \wedge C = C' \text{ then } M^\sharp(\mathrm{m}')[\mathsf{post}] \sqcap T^\sharp(\mathrm{m},i) \sqsubseteq T^\sharp(\mathrm{m},i+1) \\ \text{else } T^\sharp(\mathrm{m},i) \sqsubseteq T^\sharp(\mathrm{m},i+1)\end{array}\right) \\ \left(\begin{array}{l}\text{if } x = \mathbf{this} \wedge C = C' \text{ then } T^\sharp(\mathrm{m},i) \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{this}] \\ \text{else } \rho(C', [\![x]\!]^\sharp) \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{this}]\end{array}\right) \\ [\![x]\!]^\sharp :: [\![e_1]\!]^\sharp :: \cdots :: [\![e_j]\!]^\sharp \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{args}]\end{array}}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m},i) \ : \ x.\text{ms}(e_1, \ldots, e_j)} \tag{6}$$

$$\frac{\begin{array}{c}\forall \mathrm{m}, \mathrm{m}', \text{overrides}(\mathrm{m}', \mathrm{m}) \Rightarrow M^\sharp(\mathrm{m})[\mathsf{args}] \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{args}] \\ \forall \mathrm{m}, \mathrm{m}', \text{overrides}(\mathrm{m}', \mathrm{m}) \Rightarrow \top_{\text{class}(\mathrm{m})} \sqsubseteq M^\sharp(\mathrm{m})[\mathsf{post}] \\ \forall \mathrm{m}, \mathrm{m}', \text{overrides}(\mathrm{m}', \mathrm{m}) \Rightarrow \top_{\text{class}(\mathrm{m}')} \sqsubseteq M^\sharp(\mathrm{m}')[\mathsf{this}] \\ \forall \mathrm{m},\ M^\sharp(\mathrm{m})[\mathsf{this}] \sqsubseteq T^\sharp(\mathrm{m},0) \qquad \forall \mathrm{m},\ M^\sharp(\mathrm{m})[\mathsf{args}] \sqsubseteq L^\sharp(\mathrm{m},0) \\ \top_{\text{class}(\mathbf{main})} \sqsubseteq T^\sharp(\mathbf{main},0) \qquad Raw^- \sqsubseteq L^\sharp(\mathbf{main},0)(\mathbf{this}) \\ \forall \mathrm{m},\ \forall i,\ M^\sharp, H^\sharp, T^\sharp, L^\sharp \models (\mathrm{m},i) \ : \ P_\mathrm{m}[i]\end{array}}{M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P} \tag{7}$$

Auxiliary Operators

$$\begin{aligned}\rho(C, \text{NotNull}) &= \bot_C \\ \rho(C, \text{Raw}(X)) &= \text{if } X \preceq C \text{ then } \bot_C \text{ else } \top_C \\ \rho(C, \text{MayBeNull}) &= \rho(C, Raw^-) = \top_C\end{aligned}$$

$$[\![x]\!]^\sharp_{C,H^\sharp,t^\sharp,l^\sharp} = \begin{cases}\text{Raw}(C) & \text{if } x = \mathbf{this} \wedge t^\sharp = \bot_c \wedge l^\sharp(x) = \text{Raw}(\text{super}(C)) \\ L^\sharp(x) & \text{otherwise}\end{cases}$$

$$[\![e.f]\!]^\sharp_{C,H^\sharp,t^\sharp,l^\sharp} = \begin{cases}H^\sharp(f) & \text{if } [\![e]\!]^\sharp_{C,H^\sharp,t^\sharp,l^\sharp} = \text{NotNull} \\ & \text{or } (e = \mathbf{this} \wedge t^\sharp(f) = \text{Def} \wedge \text{class}(f) = C) \\ & \text{or } [\![e]\!]^\sharp_{C,H^\sharp,t^\sharp,l^\sharp} = \text{Raw}(X) \text{ with } X \preceq \text{class}(f) \\ \text{MayBeNull} & \text{otherwise}\end{cases}$$

Fig. 3. Analysis specification

$M^\sharp(\mathrm{m}')[\mathsf{this}]$. The constraints on $T^\sharp(\mathbf{main}, 0)$ and $L^\sharp(\mathbf{main}, 0)(\mathbf{this})$ ensure a correct initialisation of the **main** method.

Finally an abstract state is said safe (noted $\mathrm{safe}^\sharp_P(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$) when for each program point (m, i), if $P_\mathrm{m}[i]$ is of the form $x.f \leftarrow e$ or $x.\mathrm{ms}(\ldots)$ then $L^\sharp(\mathrm{m}, i)(x) \neq \mathrm{MayBeNull}$, and for all expressions e which appear in the instruction $P_\mathrm{m}[i]$, any dereferenced sub-expression e' has an abstract evaluation strictly lower than MayBeNull.

The analysis uses a function ρ to transfer information from domain $\mathrm{Val}^\sharp$ to domain $\mathrm{TVal}^\sharp$: ρ transforms an abstract reference $\mathrm{Val}^\sharp$ to a $\mathrm{TVal}^\sharp$ abstraction of a current object `this` in the class C. The notations $\bot_C$, respectively $\top_C$, correspond to the function that maps all fields defined in C to Def, respectively UnDef. The analysis also relies on an abstract evaluation $[\![e]\!]^\sharp_{C,H^\sharp,t^\sharp,l^\sharp}$ of expressions parameterised by the current class C, an abstract heap $H^\sharp$, an abstraction $t^\sharp$ of the fields (declared in class C) of the **this** object and an abstraction $l^\sharp$ of the local variables. The first equation states that the type of a local variable is obtained from the $L^\sharp$ function and can be refined from $\mathrm{Raw}(\mathrm{super}(C))$ to $\mathrm{Raw}(C)$ if **this** is sufficiently initialized. The second equation states that the type of a field is obtained from the heap abstraction if the type of the reference is NotNull or if it is a reference to an object sufficiently deeply initialized. Otherwise, it is MayBeNull. In the inference rules we write $[\![e]\!]^\sharp$ for $[\![e]\!]^\sharp_{\mathrm{class(m)},H^\sharp,T^\sharp(\mathrm{m},i),L^\sharp(\mathrm{m},i)}$

3.3 Example

The Java source code of our example is provided in Fig. 4, while Fig. 5 shows the code in the syntax defined in Sect. 2 and the constraints obtained from the rules defined in Sect. 3.2. This example is fairly simple and, for conciseness, the code of the `main` method has been omitted, but the generated constraints should be sufficient to give an idea of how the inference works.

The labels in the source starting with the character `@` are the annotations inferred by the analysis for the fields and the methods signatures. An annotation in

```
class A {
    private Object f;
    private Object g;

    A(Object o){f = o;}

    void m(){g=f;}

    public static void main(String args[]){
    Object o = new Object();
    A a = new A(o);
    a.m();
    }
}
```

Fig. 4. Java source

```
 1: class A {
 2:    Object @NotNull f;
 3:    Object @MayBeNull g;
 4:
 5:    @RawTop void init(Object @NotNull o){
 6:    M♯(A.init)[args] ⊑ L♯(A.init, 0)
 7:    M♯(A.init)[this] ⊑ T♯(A.init, 0)
 8:       0: this.{|Object,init,[] -> void|}()
 9:          ρ(Object, [[this]]) ⊑ M♯(Object.init)[this]
10:          [] ⊑ M♯(Object.init)[args]
11:          L♯(A.init, 0)[this ↦ Raw(Object)] ⊑ L♯(A.init, 1)
12:          T♯(A.init, 0) ⊑ T♯(A.init, 1)
13:       1: this.f = o
14:          [[o]] ⊑ H♯(A.f)
15:          T♯(A.init, 1)[A.f ↦ Def] ⊑ T♯(A.init, 2)
16:          L♯(A.init, 1) ⊑ L♯(A.init, 2)
17:       2: return
18:          if T♯(A.init, 2)(A.f) = UnDef then MayBeNull ⊑ H♯(A.f)
19:          if T♯(A.init, 2)(A.g) = UnDef then MayBeNull ⊑ H♯(A.g)
20:          T♯(A.init, 2) ⊑ M♯(A.init)[post]
21:    }
22:
23:    @NotNull void m(){
24:    M♯(A.m)[args] ⊑ L♯(A.m, 0)
25:    M♯(A.m)[this] ⊑ T♯(A.m, 0)
26:       0: this.g = this.f
27:          [[this.f]] ⊑ H♯(A.g)
28:          T♯(A.m, 0)[A.g ↦ Def] ⊑ T♯(A.m, 1)
29:          L♯(A.m, 0) ⊑ L♯(A.m, 1)
30:       1: return
31:          T♯(A.m, 1) ⊑ M♯(A.m)[post]
     }

  }
```

Fig. 5. Code with constraints

front of a method corresponds to the property of **this** before the invocations of the method. The other annotations are placed just before the variable they refer to.

Lines 6, 7, 24 and 25 list the constraints obtained from rule (7) in Fig. 3. They "initialize" the flow-sensitive abstractions $L^\sharp(\mathrm{m}, 0)$ and $T^\sharp(\mathrm{m}, 0)$ with the information of the method signatures. All other constraints are directly deduced from the rule corresponding to the instruction where conditional constraints have been simplified where it was possible. Lines 9 to 12 list the constraints obtained from rule (6) when $\mathrm{name}(\mathrm{m}') = \mathbf{init}$ and $\mathrm{class}(\mathrm{m}) \neq \mathrm{class}(\mathrm{m}')$. Lines 14 to 16 correspond to a field assignment on **this** of a field defined in the current class. Lines 18 to 20 correspond to a **return** instruction of a constructor, which adds the value MayBeNull in the abstract heap for all (maybe) undefined fields, while line 31 corresponds to a **return** instruction of a non-constructor method.

The methods are called from the `main` method of Fig. 4. The method `init` is annotated with Raw^- (`@RawTop`) as it is called on a completely uninitialized value. The constraint line 11 then refine Raw^- to Raw(*Object*) so if a field of `Object` were accessed in the rest of the method, the abstraction of the heap would be used. The NotNull (`@NotNull`) annotation before the argument of the method `init` is first transferred to a local variable at line 6 and then moved from o to $H^\sharp$(A.f) at line 14. At the same time, line 15 records that the field `f` is defined. Then, line 18, as `f` has been defined $H^\sharp$(A.f) is not modified whereas, line 19, $T^\sharp(\text{A.init}, 2)(\text{A.g}) = \text{UnDef}$ so $H^\sharp$(A.g) is constrained to MayBeNull.

4 Correctness

In this section we prove the correctness of the analysis. We first define (see Fig. 6) the logical link between concrete and abstract domains via a correctness relation. Then we prove (Theorem 1) that any solution of the constraint system which verifies the predicate $\text{safe}_p^\sharp$ enforces the null-pointer error safety property for the set of preconditions inferred by the analysis. The result mainly relies on a suitable subject reduction lemma (Lemma 1). The theorem has been mechanically proved with the Coq proof assistant[6].

$$\frac{v \in \text{Val}}{\text{Def} \sim (v, \mathbf{def})} \qquad \frac{v \in \text{Val} \quad d \in \{\mathbf{def}, \mathbf{undef}\}}{\text{UnDef} \sim (v, d)}$$

$$\frac{v \in \text{dom}(h) \quad \forall f \in \text{dom}(h(v)), \text{IsDef}(h(v)(f))}{\text{NotNull} \sim_h v} \qquad \frac{v \in \text{Val}}{\text{MayBeNull} \sim_h v}$$

$$\frac{v \in \text{dom}(h) \quad \forall f \in \bigcup_{A \preceq C} \text{fields}(C) \cap \text{dom}(h(v)), \text{IsDef}(h(v)(f))}{\text{Raw}(A) \sim_h v} \qquad \frac{v \neq \mathbf{null}}{Raw^- \sim_h v}$$

$$\frac{\forall x, l(x) = \bot \vee L^\sharp(x) \sim_h l(x)}{L^\sharp \sim_h l} \qquad \frac{r \in \text{dom}(h) \quad \text{fields}(C) \subseteq \text{dom}(T^\sharp) \cap \text{dom}(h(r)) \quad \forall f \in \text{fields}(C),\ T^\sharp(f) \sim h(r)(f)}{T^\sharp \sim_{h,C} r}$$

$$\frac{\forall r \in \text{dom}(h),\ \forall f \in \text{dom}(h(r)),\ h(r)(f) = (v, d) \Rightarrow d = \mathbf{undef} \vee H^\sharp(f) \sim_h v}{H^\sharp \sim h}$$

$$\frac{o = l(\mathbf{this}) \quad L^\sharp(\text{m}, i) \sim_h l \quad T^\sharp(\text{m}, i) \sim_{h,\text{class(m)}} l(\mathbf{this}) \quad H^\sharp \sim h}{(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \sim_{\text{m},o} \langle i, l, h \rangle}$$

$$\frac{M^\sharp(\text{m})[\mathsf{post}] \sim_{h,\text{class(m)}} o \quad H^\sharp \sim h}{(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \sim_{\text{m},o} h} \qquad \frac{}{(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \sim_{\text{m},o} \Omega}$$

Fig. 6. Correctness relations

An abstract element $(M^\sharp, H^\sharp)$ induces a set of method preconditions defined by

$$\text{Pre}(M^\sharp, H^\sharp)(\text{m}) = \left\{ (l, h) \;\middle|\; \begin{array}{l} H^\sharp \sim h,\ M^\sharp(\text{m})[\mathsf{this}] \sim_{\text{class(m)},h} l(\mathbf{this}), \\ V_0^\sharp \sim_h l(\mathbf{this}) \text{ and } \forall i = 1..n, V_i^\sharp \sim_h l(i) \end{array} \right\}$$
$$\text{with } M^\sharp(\text{m})[\mathsf{args}] = V_0^\sharp :: \cdots :: V_n^\sharp$$

[6] The proof is available at `http://www.irisa.fr/lande/pichardie/np/`

Note that a method m which is not called by any other method in the program, will be associated with a non-null assumption on all its parameters. This corresponds to the *non-null by default* approach advocated by Chalin and James [2].

We write $\rightarrow_{\mathrm{m}}^{(n)}$ (resp $\rightarrow_{\mathrm{m}}^{(n)*}$) for the operational step relation (resp. transitive closure) where exactly n method calls are performed during step (resp. transitive closure), including sub-calls.

Lemma 1 (Subject reduction). *Let* $(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \in State^\sharp$ *such that* $M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P$ *and* $safe_P^\sharp(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$. *Let* n *be an integer such that for all* k, $k < n$, *and all methods* m, *local variables* l, *heap* h *and configuration* X *if* $(l, h) \in Pre(M^\sharp, H^\sharp)(m)$ *and* $\langle 0, l, h\rangle \rightarrow_m^{(k)*} X$ *then* $(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \sim_{m,l(\textbf{\textit{this}})} X$ *and* $X \neq \Omega$.

Let p *be an integer,* i *a program counter,* l *local variables,* h *a heap and* X *a configuration such that* $\langle i, l, h\rangle \rightarrow_m^{(p)} X$, $(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \sim_{m,o} \langle i, l, h\rangle$ *and* $p \leq n$ *then* $(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \sim_{m,o} X$ *and* $X \neq \Omega$.

Proof. See Appendix C in [12]

Theorem 1 (constraint system soundness). *If there exists* $(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$ *such that* $M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P$ *and* $safe_P^\sharp(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$ *holds then* P *is null-pointer error safe.*

Proof. The proof proceeds by induction on the maximum number of method calls in the execution sequences and then another induction on the length of intra-method derivation, in order to be able to conclude with the subject reduction Lemma 1.

5 Inference

Expressing the analysis in terms of constraints over lattices has the immediate advantage that inference can be obtained from standard iterative constraint solving techniques for static analyses. Proposition 1 asserts that there is a decision procedure to detect programs which are verifiable with the analysis.

Proposition 1. *For all program* P *there exists an algorithm which decides if there exists* $(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \in State^\sharp$ *such that* $M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P$ *and* $safe_P^\sharp$ $(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$.

Proof. It is a standard proof since the system of constraint is composed of monotone functions on a finite lattice. The least solution can then be computed and be checked with respect to predicate $safe_P^\sharp$.

Furthermore, the present analysis is modular in the sense that, rather than performing an analysis of all classes of a program, it is possible to describe certain classes (*e.g.*, classes coming from a library) by providing *interfaces* consisting of some method signatures $M^\sharp(\mathrm{m})$, $\mathrm{m} \in M^{\mathrm{fix}}$ and field invariants $H^\sharp(f)$, $f \in F^{\mathrm{fix}}$

relative to the classes.[7] The validity of these signatures can be established once and for all by the modular type checker proposed by Fähndrich and Leino (which works class by class). The analysis of partial programs would then proceed in several stages. First, the constraint system is generated only for the available classes. The system may refer to the variables $M^\sharp(\mathrm{m})$, $\mathrm{m} \in M^{\mathrm{fix}}$ and $H^\sharp(f)$, $f \in F^{\mathrm{fix}}$. Then the partial system is solved starting the iteration *à la* Kleene from

$$(M_0^\sharp, H_0^\sharp, T_0^\sharp, L_0^\sharp) = (\bot \sqcup \{\mathrm{m} \mapsto M^\sharp(\mathrm{m})\}_{\mathrm{m} \in M^{\mathrm{fix}}}, \bot \sqcup \{f \mapsto H^\sharp(f)\}_{f \in F^{\mathrm{fix}}}, \bot, \bot)$$

If one of the variables $M^\sharp(\mathrm{m})$, $\mathrm{m} \in M^{\mathrm{fix}}$ or $H^\sharp(f)$, $f \in F^{\mathrm{fix}}$ has to be updated during the iteration, the constraint resolution fails, since this means there does not exist a solution compatible with the proposed signatures.

Theorem 2 (Relative completeness of the modular solver). *If the previous algorithm halts then the partial constraint system has no solution compatible with the signatures provided for the unknown classes.*

Proof. Let S denote the set

$$\{(M^\sharp, H^\sharp, T^\sharp, L^\sharp) \mid M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P \wedge (M_0^\sharp, H_0^\sharp, T_0^\sharp, L_0^\sharp) \sqsubseteq (M^\sharp, H^\sharp, T^\sharp, L^\sharp)\} .$$

Suppose the previous algorithm halts. Since the iteration is ascending it means the least solution of S is necessarily strictly greater than $(M_0^\sharp, H_0^\sharp, T_0^\sharp, L_0^\sharp)$. Since any solution compatible with the signatures $M^\sharp(\mathrm{m})$, $\mathrm{m} \in M^{\mathrm{fix}}$ or $H^\sharp(f)$, $f \in F^{\mathrm{fix}}$ is in S, we can conclude that such a solution does not exist.

6 Fähndrich and Leino's Type System

In this section we compare our analysis with the type system proposed by Fähndrich and Leino [6]. This comparison relies on a formal definition[8] of their notion of typable program. For conciseness, the full type system is not included here but can be found in [12].

A typing judgment for expression e is of the form $\Gamma, L \vdash e : \tau$ with Γ a type annotation for fields and methods, L a type annotation for local variables and τ a type in $\mathrm{Val}^\sharp$. Each instruction instr is also associated to a type judgement $\Gamma, \mathrm{m} \vdash \mathrm{instr} : L \to L'$ where m is the current method, L the current annotation for local variables and L' a valid annotation after the execution of instr. [`inits` F] is a method annotation that indicates the method initializes the fields in the set F. A program is said *well-typed* w.r.t. Γ if there are no overridden methods annotated as [`inits` F], arguments are contravariant and there exists L such that for all methods m and for all program points i, either $P_{\mathrm{m}}[i] =$ **return**

[7] Modular inference is less precise: to keep the same precision the analysis would need richer annotations for the libraries.

[8] Note that in their paper the authors only propose an informal definition so what we formalise here is only our interpretation of their work.

or for all successors points j of i, their exists L' such that $L' \sqsubseteq L(\mathrm{m}, j)$ and $\Gamma, \mathrm{m} \vdash P_{\mathrm{m}}[i] \ : \ L(\mathrm{m}, i) \to L'$.

Figure 7 shows, as an example, the judgment for field assignments. It checks two properties: 1) the type $\Gamma[f]$ of the field f subsumes the type τ of the expression e and 2) the type of the reference x is not MayBeNull.

$$\frac{\Gamma, L \vdash e : \tau \qquad \tau \sqsubseteq \Gamma[f] \qquad L(x) \neq \text{MayBeNull}}{\Gamma, \mathrm{m} \vdash x.f \leftarrow e \ : \ L \to L}$$

Fig. 7. Typing judgment for field assignments

This type system is coupled with a data flow analysis to ensure that all fields not declared as MayBeNull in class C are sure to be defined at the end of all constructors of C. It is a standard data flow analysis, the full description can be found in [12]. $\mathrm{F}(\mathrm{m}, i)$ represents the fields that are not initialized at program point (m, i). At the beginning of every constructor m, it is constrained to the set of fields defined in the class of m. For field assignation on **this**, the field is not propagated to the next node. For method calls to methods tagged as [`inits` F], fields in F are not propagated to the next node, but the set of undefined fields is propagated to the first instruction of the callee. It is then checked that those tagged methods initialize their fields.

Figure 8 shows an extract of the data flow analysis. It distinguishes two cases depending on whether the field assignation is on **this**, in which case the field assigned is removed from the set of undefined fields, or on another object, in which case the set of undefined fields is propagated.

$$\frac{\mathrm{F}(\mathrm{m}, i) \setminus \{f\} \subseteq \mathrm{F}(\mathrm{m}, i+1)}{\Gamma, \mathrm{F} \models (\mathrm{m}, i) : \mathbf{this}.f \leftarrow e} \qquad \frac{x \neq \mathbf{this} \qquad \mathrm{F}(\mathrm{m}, i) \subseteq \mathrm{F}(\mathrm{m}, i+1)}{\Gamma, \mathrm{F} \models (\mathrm{m}, i) : x.f \leftarrow e}$$

Fig. 8. Data flow rule for field assignments

Theorem 3. *If P is FL-typable then there exists $(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$ such that $M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P$ and $safe^\sharp_P(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$ holds.*

Proof. We show that if P is FL-typable for a given Γ and L then this type annotations represent a valid solution of the constraint system which furthermore satisfies the $\mathrm{safe}^\sharp_P$ property.

Corollary 1 (FL type system soundness). *If there exists Γ such that P is FL-typable then P is null-pointer error safe w.r.t. the preconditions given by Γ.*

Proof. Direct consequence of Theorem 3 and Theorem 1.

Theorem 4. *There exists P such that P is not FL-typable and there exists $(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$ such that $M^\sharp, H^\sharp, T^\sharp, L^\sharp \models P$ and $safe^\sharp_P(M^\sharp, H^\sharp, T^\sharp, L^\sharp)$ holds.*

Proof. As shown in Fig. 3, the analysis of expressions is parametrized on the abstraction of `this` in order to know if a field has already been defined or not. In Fähndrich and Leino's analysis, the type checking is separated from the data-flow analysis that knows which fields have already been defined. For example, in

```
class C {
  Object f; //NotNull
  Object g; //MayBeNull or NotNull?
  public C(){ this.f = new Object(); this.g = this.f;}
}
```

our analysis benefits from the abstraction of `this` and knows `this.f` has been initialized before it is assigned to `this.g`. In Fähndrich and Leino's analysis, an intermediate local variable set to the new object and affected to `f` and `g` or an explicit cast operator which checks the initialization at run-time would be needed in order to type the program. Our abstraction of `this` can also be passed from a method to another, here also keeping some more information as the intra-procedural data-flow of Fähndrich and Leino. In the Soot suite we have analyzed [16] (see the next section), 5% of the constructors use fields that have been just defined.

7 Towards a Null-Pointer Analyzer for Java Bytecode

A prototype of the analysis has been implemented. The analysis works at the Java bytecode (JVML) level but annotations on fields and method can be propagated at the source level with a minor effort.

Working in the fragment of JVML without exceptions does not modify the analysis but some points need to be taken in account. JVML is a **stack language**, we therefore need to track aliases of `this` on the stack to know when a field manipulation or a method call is actually done on `this`. Operations on **numbers** do not add any difficulty as JVML is typed and numbers are guarantied not to be assigned to references. The **multi-threading** cannot interfere with weak updates, but it could interfere with the strong updates used on the abstraction of `this`. However, as the initialization evolves monotonically (*i.e.* a field once defined cannot be reverted to undefined) it is still correct. **Static methods** do not have an abstraction of `this` and do not add any other difficulty to handle.

Static fields are difficult to analyse precisely as they are initialized through `<clinit>` methods. **Exceptions** in Java can be Raw object, *i.e.* it is possible in a constructor to throw **this** as an exception. We could imagine a solution where abstract method signatures would include the type of the exceptions a method can throw. Tracking initialization of **array** cells precisely is also challenging as checking that all cells have been initialized requires numerical abstraction. We currently and conservatively annotate static fields and array cells as MayBeNull and references to caught exception objects as Raw^{-} and left their precise handling for future work.

Other small optimizations (that have not been formalized) are possible. For example, the abstraction of `this` could be done for all other references of type $\mathrm{Raw}(X)$ (or Raw^{-}). It would give a more precise information on fields in general.

The first performance tests are promising as large programs such as Javacc 4.0 and Soot 2.2.4 can be analysed on a laptop in about 40 seconds and 5 minutes respectively. The analysis still needs to be completed with a path-sensitive analysis [4] to recover non-nullness information from the conditionals.

8 Conclusions and Future Work

We have defined a semantics-based analysis and inference technique for automatically inferring non-null annotations for fields. The analysis has been proved correct and the correctness proof has been machine-checked in the proof assistant Coq. This extends and complements the seminal paper of Fähndrich and Leino in which is proposed an extended type system for verifying non-null type annotations. Fähndrich and Leino's approach mixes type system and data-flow analysis. In our work, we follow an abstract interpretation methodology to gain strong semantic foundations and a goal-directed inference mechanism to find a minimal (*i.e.* principal) non-null annotation. By the same token we also gained in precision thanks to a better communication between abstract domains. We then proved the correctness of our analysis and its completeness with respect to their type system.

Variations of the present analysis can be envisaged. For example, in our analysis, preconditions for methods are computed as the least upper bounds of the conditions verified at call points. Another approach would be to infer the weakest preconditions that prevents null-pointer exceptions.

In the future, we also plan to extend our analysis and its correctness proof to the full Java bytecode language. To manage this substantial extension it is important to be able to machine-check the correctness proof, which will necessarily be large. Our previous experience with developing a certified static analyser for Java [1] using the Coq proof assistant leads us to believe that such a formalisation is indeed feasible.

References

1. Cachera, D., Jensen, T.P., Pichardie, D., Rusu, V.: Extracting a data flow analyser in constructive logic. Theoretical Computer Science 342(1), 56–78 (2005)
2. Chalin, P., James, P.R.: Non-null references by default in Java: Alleviating the nullity annotation burden. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 227–247. Springer, Heidelberg (2007)
3. Cielecki, M., Fulara, J., Jakubczyk, K., Jancewicz, Ł.: Propagation of JML non-null annotations in Java programs. In: Proc. of the 4th international symposium on Principles and practice of programming in Java (PPPJ 2006), pp. 135–140. ACM Press, New York (2006)
4. Das, M., Lerner, S., Seigle, M.: Esp: path-sensitive program verification in polynomial time. In: Proc. of the Conference on Programming language design and implementation (PLDI 2002), pp. 57–68. ACM Press, New York (2002)

5. Ekman, T., Hedin, G.: Pluggable non-null types for Java (ch. V). In: Ekman, T. (ed.) Extensible Compiler Construction, June 2006, Lund University (2006)
6. Fähndrich, M., Leino, K.R.M.: Declaring and checking non-null types in an object-oriented language. In: Proc. of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003), pp. 302–312. Springer, Heidelberg (2003)
7. Fähndrich, M., Xia, S.: Establishing object invariants with delayed types. In: OOPSLA 2007: Proc. of the 22nd conference on Object Oriented Programming Systems and Applications, pp. 337–350. ACM, New York (2007)
8. 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)
9. Freund, S.N., Mitchell, J.C.: A formal framework for the java bytecode language and verifier. In: Proc. of the 14th conference on Object-oriented programming, systems, languages, and applications (OOPSLA 1999), pp. 147–166. ACM Press, New York (1999)
10. Hovemeyer, D., Pugh, W.: Finding more null pointer bugs, but not too many. In: PASTE 2007: Proc. of the 7th workshop on Program analysis for software tools and engineering, pp. 9–14. ACM Press, New York (2007)
11. Hovemeyer, D., Spacco, J., Pugh, W.: Evaluating and tuning a static analysis to find null pointer bugs. SIGSOFT Softw. Eng. Notes 31(1), 13–19 (2006)
12. Hubert, L., Jensen, T., Pichardie, D.: Semantic foundations and inference of non-null annotations. Research Report 6482, INRIA (March 2008)
13. Kawahito, M., Komatsu, H., Nakatani, T.: Effective null pointer check elimination utilizing hardware trap. SIGPLAN Not. 35(11), 139–149 (2000)
14. Leino, K.R.M., Saxe, J.B., Stata, R.: ESC/Java user's manual. Compaq Systems Research Center, technical note 2000-002 edition (October 2000)
15. Male, C., Pearce, D.J., Potanin, A., Dymnikov, C.: Java bytecode verification for @NonNull types. In: Proc. of the Conference on Compiler Construction (CC 2008), Springer, Heidelberg (2008)
16. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot — a Java bytecode optimization framework. In: CASCON 1999: Proc. of the 1999 conference of the Centre for Advanced Studies on Collaborative research, p. 13. IBM Press (1999)

Redesign of the LMST Wireless Sensor Protocol through Formal Modeling and Statistical Model Checking

Michael Katelman, José Meseguer, and Jennifer Hou

Department of Computer Science
University of Illinois at Urbana-Champaign, U.S.A.
{katelman,meseguer,jhou}@uiuc.edu

Abstract. The local minimum spanning tree (LMST) topology control protocol tries to maintain connectivity in an ad-hoc wireless sensor network while minimizing power consumption and maximizing data bandwidth. Our formal, statistical model checking analysis of LMST under realistic deployment conditions shows that the invariant of maintaining network connectivity is easily lost. We then propose a formally-based system redesign methodology in which quantitative temporal logic formulas and further statistical model checking can be used to identify the causes of bugs, and to reach a correct system redesign. We show this methodology effective in the redesign of a version of LMST that ensures network connectivity under realistic deployment conditions.

1 Introduction

The design of wireless sensor network protocols presents many challenges. On the one hand, it is infeasible to comprehensively evaluate an ad-hoc wireless sensor network protocol based solely on deployment in the field. On the other, faithfully modeling such a protocol is far from trivial, because this requires a precise model of communication in which physical distance, location, power, and time must all be taken into account. Simulation is a widely used analysis method; but it falls short of formal analysis in its capacity to verify in a more conclusive way desired requirements. Formal modeling and analysis itself is nontrivial, because of the need for faithfully capturing the communication model, real time, and the often probabilistic algorithms (e.g. 802.11 MAC contention), or probabilistic phenomena (e.g. quartz clock drift).

The best way of using such formal modeling and analysis is not *a posteriori*, after a wireless protocol has been designed, but as a powerful method to *design* and *redesign* several times such a protocol, using the insights gained from the formal analysis to meet the desired requirements in a final design. In this work we do exactly this for the *local minimum spanning tree* (LMST) topology control protocol [18]. Only a high-level design of such a protocol under idealized circumstances existed prior to our work. At that idealized level, the key property that the protocol always maintains network connectivity had been shown by mathematical analysis in [18]. The nontrivial challenge has been to refine

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 150–169, 2008.

this high-level, idealized design into an implementable protocol version that can deal in practice with unavoidable issues such as clock drift, MAC contention, and transmission delay. The challenge has been nontrivial because our formal analysis has shown that the key invariant of maintaining network connectivity fails rather badly when these additional conditions are accounted for.

Our starting point has been the work of Ölveczky and Thorvaldsen [23,24]. They show how to formally model and analyze the OGDC wireless sensor network protocol using Real-Time Maude [22], an extension of the rewriting logic language Maude [5] for real-time and hybrid systems. Real time is of the essence for wireless sensor network protocols such as OGDC and LMST; and we have adopted their elegant way of faithfully modeling all relevant aspects of a wireless communication model, such as its broadcast nature, plus its sensitivity to location, distance, and transmission range; and of specifying message sending and receiving events by rewrite rules, proposed in [23,24]. We begin by specifying in this way the idealized LMST protocol as a real-time rewrite theory and analyzing it in Real-Time Maude, thus confirming by model checking the connectivity maintenance property established analytically in [18]. This serves as our base specification and provides key infrastructure on which to tackle the important challenge of arriving at a realistic (re-)design of the LMST protocol.

As soon as we introduce into the protocol model more realistic implementation details and environmental pressures, two important things happen. First, since various probabilistic phenomena naturally appear at this more realistic level, our formal specifications of the various refinements of the original model now become real-time *probabilistic* rewrite theories [13]. Probabilistic rewrite theories can be not only simulated in Maude using standard sampling techniques [3], they can also be formally analyzed by statistical model checking using the VeStA tool [26]. Second, our analysis shows that the idealized design fails quite badly to maintain network connectivity when such realistic issues are made explicit in the model; and therefore LMST requires a nontrivial *redesign*.

This work makes two main contributions. The first is to show, using a concrete state-of-the-art wireless sensor protocol like LMST, how the very successful Real-Time Maude approach to modeling and analysis of wireless sensor protocols initiated in [23,24] can be seamlessly extended to the probabilistic setting, both at the level of specifications (passing from real-time rewrite theories to probabilistic real-time rewrite theories), and at the level of formal analysis (passing from LTL model checking in Real-Time Maude to statistical model checking in VeStA). We believe that this extension is quite useful because: (i) wireless sensor networks must operate in a probabilistic environment and often include probabilistic algorithms in some protocol components (e.g. 802.11 MAC contention); and (ii) performance issues are of the essence, and it is therefore very useful to generalize the absolute Boolean-valued guarantees of LTL requirements to probabilistic real-valued guarantees associated to probabilistic temporal logic requirements in a logic like QuaTEx [3]. In QuaTEx, the evaluation of a temporal logic formula yields a real number (not necessarily between 0 and 1) corresponding to some quantitative measurement of the system.

Our second contribution, also illustrated in the context of LMST for the sake of concreteness, but of general applicability, is to show how this style of probabilistic real-time formal specification and analysis can be the basis of a very useful *design* and *redesign methodology*. In our methodology, probabilistic real-time formal specifications and quantitative statistical model checking analysis are used throughout the design process to support three mutually-reinforcing tasks: (i) to uncover flaws in a given design; (ii) to conjecture the *causes* of the various malfunctions and to *confirm* such conjectures by means of statistical correlations between further analyses; and (iii) to then use the confirmed conjectures of the hypothesized causes of flaws to *redesign* the protocol several times and ultimately to *verify* by statistical model checking that the final design satisfies the desired requirements. In addition to LMST, the methodology is widely applicable for other wireless protocols and to other probabilistic systems, such as DoS protection protocols [2] and stochastic hybrid systems [20]. Our application of the methodology in this paper results in a new, implementable design of the LMST protocol that satisfies desired requirements in the face of realistic operating conditions.

2 The Idealized Local Minimum Spanning Tree Protocol

We begin by briefly reviewing the idealized version of the local minimum spanning tree (LMST) topology control protocol presented in [18]. The function of a topology control protocol is to define which nodes in an ad-hoc wireless sensor network communicate with each other, and with what transmission power they communicate. The goal is to minimize power consumption, prolong network lifetime, and maximize data bandwidth while maintaining network connectivity. In the case of the LMST protocol, a distributed algorithm is employed whereby each sensor node periodically updates its own *local topology*. The local topology of a node is the set of neighbors to which it routes data.

In the protocol, each wireless node is presumed to have internal quartz clock timers, a memory for buffering messages, and a wireless transmitter which is adjustable to different power levels. The periodic, real-time nature of the protocol is governed by a global constant called the *round time*, denoted rd, and is approximately $10s$. Each node constantly employs one of its timers to count the time between round boundaries, at which point the node may adjust its local topology by changing its wireless transmission strength. We refer to this timer as the *round timer*. There are therefore two notions of a round, one *global* and one *local*. A global round is any interval $[t, t + rd]$ where t is a multiple of rd. A local round is determined with respect to a particular node, and is defined as any interval between successive round timer expirations. The protocol is then defined by what happens when the local round timer of a node expires:

1. The node first broadcasts a message, called a *hello message*, at *maximum transmission strength*. The hello message contains a unique identifier of the node and its current physical location. Hello messages are buffered by any *visible neighbor*, that is, any node within wireless transmission range.

2. The node reads from its message buffer all hello messages received during the previous round and distills from these a graph of its visible neighbors weighted by distance.
3. Taking the local graph of visible neighbors just distilled by the node, it then calculates the minimum spanning tree of that graph.
4. The nodes in the local minimum spanning tree which are directly connected (one-hop away) are selected to be the node's new *neighbors*, meaning those to which it will transmit data during this local round.
5. The node resets its round timer for *rd*, and waits for the timer to expire.

As shown in [18], LMST has a number of advantageous properties, including low power usage, and a provably small number of neighbors for each node, which reduces medium contention and increases bandwidth. Furthermore, it is also shown that LMST satisfies the crucial property of *maintaining network connectivity*. That is, if the graph whose edges link the sensor nodes within wireless reach of each other is connected, then the considerably smaller subgraph computed by LMST is also connected. However, LMST is an *idealized* design, which does not take into account crucial issues that must be faced in a real implementation. As we show later, the crucial requirement of maintaining network connectivity is soon lost when such issues are modeled. It thus remains an open question how LMST can be refined into a realistic design where such realistic issues are addressed and where network connectivity is still maintained.

3 Idealized LMST Model in Real-Time Maude

This section describes our formal specification and analysis in Real-Time Maude (RTM) [22] of the idealized version of LMST summarized in Section 2. We do this following the general methodology for formal modeling of wireless sensor networks proposed in [23,24], incorporating many modeling constructions essentially unchanged. This first step of modeling and analysis serves two purposes. First, since probabilistic phenomena are not yet modeled at this idealized level, it serves as a warm-up exercise to later see how real-time specifications of wireless sensor network protocols in RTM can be naturally extended to probabilistic real-time specifications. Second, since the network connectivity invariant was only shown by high-level analytic arguments in [18] but never formally verified, it also serves as a sanity check to obtain independent, model checking evidence that the invariant holds, and to indirectly gain further confidence that our formal RTM specification faithfully captures the idealized LMST design. We first recall some background on real-time rewrite theories and their use in modeling wireless network protocols. We then summarize the specification of LMST as a real-time rewrite theory and its formal analysis in RTM.

3.1 Modeling Wireless Networks in Real-Time Maude

The key idea of rewriting logic [19] is to model a concurrent system as a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational theory whose types and

function symbols are described by a signature Σ, and having a set of equations E; and where R is a collection of rewrite rules. The basic idea is that the *states* of the concurrent system thus specified are elements of the algebraic data type (initial algebra) $T_{\Sigma/E}$ associated to the equational theory (Σ, E), whereas the *concurrent transitions* of the system are the possible rewrites allowed by the rewrite rules R. For our purposes in this paper it is useful to instantiate this general idea to the case of *object-based distributed systems*. For such systems it is often useful to model the distributed state as a "soup" or multiset of *objects* and *messages*, of types (sorts) *Object* and *Msg*. Such distributed, soupy states can be called *configurations* and belong to a sort *Configuration*, which contains *Object* and *Msg* as *subsorts*. The soup-like nature of the state is algebraically modeled by declaring a binary multiset union operator (which we can describe with empty, juxtaposition syntax) satisfying the axioms of *associativity* and *commutativity*, and having the empty multiset *null* as its *identity* element. Thus, if O_1 and O_2 are objects, and M_1, M_2, and M_3 are messages, then the juxtaposition $O_1\, O_2\, M_1\, M_2\, M_3$ is a configuration made up of those objects and messages, where the associativity and commutativity axioms mean that no parentheses are needed, and the order in which the objects and messages appear is immaterial. In the Maude rewriting logic language [5] and in its RTM extension, objects of a given class C in a given state are represented as terms of the form

$$\texttt{<}\ O : C\ \texttt{|}\, attr_1 : val_1, \ldots, attr_n : val_n \texttt{>}$$

where O is the object's name or identifier, C is the object's class, and where val_1 to val_n are the current values of the attributes $attr_1$ to $attr_n$, respectively, which in a given class C are required to have specified sorts $s_1, \ldots, s_n$. This requirement is specified in a class declaration of the form

$$\texttt{class}\ C\ \texttt{|}\ attr_1 : s_1, \ldots, attr_n : s_n.$$

The *dynamics* of an object-based distributed system are then specified by multiset rewrite rules, where one or more objects and/or messages in a configuration are rewritten to other objects and/or messages.

How about *real-time* object-based distributed systems? They can be formally specified by *real-time rewrite theories* [21], where the rewrite rules now have *time duration* information, that is, they are conditional rewrite rules of the form

$$t \stackrel{\tau}{\longrightarrow} t' \ \ \mathit{if}\ \ \mathit{cond}$$

with t and t' multiset expressions involving objects and messages, and with τ a *time* expression, that is, a term of sort *Time*, where the time domain can be chosen to be either discrete or dense. If $\tau = 0$, then we call the rule an *instantaneous* rule, and we omit the 0 label. Otherwise we call the rule a *tick* rule, because time is advanced. Since time should advance not just for a local state, but for the whole system, the global configuration of a system is encapsulated by a bracket operator, so that the global state has the form $\{t\}$, with t a configuration of objects and messages. theories can be *desugared* into ordinary rewrite theories

[22]. The trick is to model the global state $\{t\}$ as a pair $(\{t\}, \alpha)$, where α is the current value of a global clock. Then a tick rule $\{t\} \stackrel{\tau}{\longrightarrow} \{t'\}$ *if* *cond* is desugared into an ordinary rewrite rule

$$(\{t\}, \alpha) \longrightarrow (\{t'\}, \alpha + \tau) \ \ \textit{if} \ \ \textit{cond}$$

Real-Time Maude is an extension of Maude that directly supports the specification of real-time systems as real-time rewrite theories. It performs the above desugaring to execute such rewrite theories in the underlying Maude system. It also supports breadth-first search and model checking of LTL properties by compilation into Maude, where now such LTL properties may involve predicates that inspect the global clock or the state of particular timers in some objects, and where the model checking can specify a time bound [22]. In [23,24], Real-time Maude has been shown to be very well suited to model and formally analyze wireless network protocols, in particular we utilize their models of standard wireless communication types. The first is unicast messaging:

```
sort DirectedMsg .   subsort DirectedMsg < Configuration .
op directed-msg : Receiver Msg -> DirectedMsg .
```

Broadcast messages are used generally to send a message to all nodes within a certain physical radius of the transmitting node. In our specification we use them to model both wireless data transmission as well as part of our 802.11 MAC model. They use the following syntax:

```
sort BroadcastMsg .  subsort BroadcastMsg < Configuration .
op broadcast-msg : Sender Msg -> BroadcastMsg .
```

Message broadcasting is modeled as in [24], by defining a set of equations to turn each broadcast message into a set of unicast messages, one such message for each node within transmission range.

The last wrapper is used to delay the reception of a message with respect to real time. If the message is a true wireless transmission, then this delay corresponds to the transmission delay; in other cases it may model different things. We use delayed messages extensively to model timers like the one used by each node to count time between successive rounds.

```
sort DelayedMsg    .  subsort DelayedMsg    < Configuration    .
sort Delay         .  subsort TimeInf       < Delay            .
op delayed-msg : DirectedMsg  Delay -> DelayedMsg [right id: 0] .
op delayed-msg : BroadcastMsg Delay -> DelayedMsg [right id: 0] .
```

Finally, [22] shows how all of the time-elapsing events of the system should be distilled into a single tick rewrite rule of the form

```
crl [tick] :
   {Cx:Configuration}  => {delta(Cx, Tx)} in time Tx:Time
if Tx <= mte(Cx) .
```

using two operator symbols `delta` and `mte`. The first operator defines the effect of the time elapse for the system configuration, and the second operator defines the maximum time that can elapse before the next instantaneous event. Both are defined over the *Configuration* sort, and act to distribute over the objects and messages in the configuration. For example, the `delta` function is partially defined by (we indicate the sort of each variable with its first use)

```
eq delta(delayed-msg(DMx:DirectedMsg, Dx:Delay)
      Cx:Configuration
      , Tx:Time)
=  delayed-msg(DMx, Dx monus Tx)
   delta(Cx, Tx) .
```

3.2 Idealized Model of LMST in Real-Time Maude

Our definition of the LMST protocol hand specifies the description given in [18] through a set of (mostly) localized rewrite rules governing state changes of the nodes individually. In order to make our version conform to the idealized definition of the protocol, we have to define guards for each rewrite rule that consider the entire state before firing. For example, we guard the rule for updating a node's local topology to ensure that all outstanding hello messages are received prior to doing the update calculation. This corresponds to a tacit assumption made by the protocol that before a topology update occurs, a hello message has been received from every visible neighbor.

The protocol is idealized in the sense that it is free from real-world conditions such as message loss, node mobility, imperfect quartz clock timers, and so on. In addition, we assume that all the round timers are synchronized, so that all of them always signal a new round in unison. The following class definition defines wireless sensor nodes:

```
class SensorNode |
   location          : Location
   , received-HMs    : HelloMsgValueSet
   , neighbor-set    : NodeIDSet
   , transmit-radius : Float .
```

The attributes give the node's physical location, the buffered hello messages received since the start of the current round, the current set of neighbors, and the current transmission radius. Since the goal of a topology control protocol is to determine each sensor node's wireless transmission strength, it may seem curious that this information is missing from the above definition. The reason is that our analysis will only be concerned with connectivity, for which having the transmission radius is simply more convenient. We note that transmission strength can be calculated from the transmission radius and knowledge about the radio propagation model.

There are three message constructors: one to represent wireless communication of hello messages, and two others representing other state-changing actions that a node must take itself.

```
msg round-timer-msg  :                  -> Msg .
msg update-msg       :                  -> Msg .
msg hm-msg           : HelloMsgValue    -> Msg .

op <_,_> : NodeID Location -> HelloMsgValue .
```

The payload of a hello message is the sending node's identifier plus its location. This message is broadcast by each node at the beginning of each new round (see Section 2). There are exactly three rewrite rules in the model, one for each of the messages above. The idea is that every concurrent event in the system has an associated message, and is acted upon when received by the node it is directed to. This idea is similar to what is suggested in [3].

Instead of presenting the rules verbatim from our idealized model, we simply indicate what each rule does using prose. This saves space and omits syntactic overhead which does not directly contribute to the interesting actions being performed. Our model, which can be downloaded [1], contains the exact definitions.

The events associated with the above messages are as follows:

1. When a `round-timer-msg` is consumed, the associated node performs three actions. It first broadcasts a hello message identifying itself and giving its current location. Second, it schedules an `update-msg` for itself. Third, it resets its round timer, emitting a delayed `round-timer-msg`.
2. When a `hm-msg` is consumed, the receiving node simply adds the message payload to the `received-HMs` set.
3. When a `update-msg` is consumed, the receiving node updates its local topology (setting `neighbor-set`). The updated topology is calculated from the minimum spanning tree of the graph defined by the hello messages received during the previous round. One slight complication is that for the topology to conform to the idealized specification, we must ensure that all hello messages be received during the current time instant before a topology update is executed. This is checked for using an equationally defined guard and a conditional rewrite rule.

The three rules described above essentially define the entire model. Most of the specification focuses on defining the minimum spanning tree calculation.

The formal model can be analyzed through *time-bounded LTL model checking* in RTM. When we analyze the model, we start with an initial configuration that is primed by inserting one directed round timer message for each node in the configuration, set to be consumed instantly at time 0. The property that we are interested in is *total network connectivity*, which means that multi-hop connections exist from every node to every other node in the network. As an LTL safety formula, the property is expressed as the invariant $\Box$ `connected`, with `connected` an equationally-defined predicate (see [1]). It computes the strongly connected components of the graph defined by the union of all neighbor sets.

The model checking that we have done is to check that the connectedness property holds in our model *with respect to a set of randomly selected initial configurations*, each involving a small number (4) of nodes and with a time bound

of one round. It is important to make clear exactly what this result means with respect to the idealized protocol. The non-determinism exhibited in our model comes from selecting the order in which nodes broadcast hello messages, receive messages, *etc.* However, due to the guard on the topology update rule, the non-determinism that was added does not result in many interesting interactions.

4 Probabilistic Modeling and Analysis of LMST

A major issue with the analysis of the previous section is that standard model checking algorithms do not apply to *probabilistic phenomena*, such as the uniform distribution of nodes in a sensing area. Indeed, many probabilistic phenomena naturally exist for wireless protocols. In this section we describe two realistic refinements of the idealized LMST model: one with unsynchronized local round timers, and the other with quartz clock drift, 802.11 MAC contention, and message delay. Due to space limitations, we have omitted two other refinements – one with node mobility and another with probabilistic message loss – that we have also modeled and analyzed (see [11]). The refinements are modified versions of the idealized model, transformed into a standard rewrite theory, with new rules and probabilistic annotations. The annotations take the formalization outside the realm of standard rewrite theories and into the realm of *probabilistic rewrite theories* [13,3]. After the probabilistic models are defined, we show how to automatically analyze them using the *logic of quantitative temporal expressions* (QuaTEx) [3] and the statistical model checking algorithms implemented by the VeStA tool [3,26]. The analysis reveals significant disconnectedness, or *bugs*, under the conditions imposed by each of the refinements.

4.1 Probabilistic Rewrite Theories, QuaTEx, and VeStA

A *probabilistic rewrite theory* [13,3] replaces the usual rewrite rules with probabilistic ones of the form

$$l(\boldsymbol{x}) \longrightarrow r(\boldsymbol{x}, \boldsymbol{y}) \texttt{ with probability } \boldsymbol{y} := p(\boldsymbol{x})$$

Such a rule is *non-deterministic*, because the term r has new variables $\boldsymbol{y}$ disjoint from the variables $\boldsymbol{x}$ appearing in l. Therefore, a substitution θ for the variables $\boldsymbol{x}$ appearing in l that matches a subterm of a term t at position q does not uniquely determine the next state after the rewrite: there can be many different choices for the next state, depending on how we instantiate the extra variables $\boldsymbol{y}$ in r. In fact, we can denote the different next states by expressions of the form $t[r(\theta(\boldsymbol{x}), \sigma(\boldsymbol{y}))]_q$, where θ is fixed as the given matching substitution, but σ ranges over all possible substitutions for the new variables $\boldsymbol{y}$. The probabilistic nature of the rule is expressed by the notation: $\texttt{with probability } \boldsymbol{y} := p(\boldsymbol{x})$, where $p(\boldsymbol{x})$ is a probability measure on the set of substitutions σ (modulo the equations E in a given rewrite theory). However, the probability measure $p(\boldsymbol{x})$ *may depend on the matching substitution* θ. We sample $\boldsymbol{y}$, that is, the substitution σ, probabilistically according the probability measure $p(\theta(\boldsymbol{x}))$.

PMaude [3] is an extension of Maude for probabilistic rewrite theories. Like Real-Time Maude, PMaude is implemented as a theory transformation using Maude's reflection capabilities. This involves desugaring the probability annotation in a probabilistic rewrite rule and giving rewriting definitions for the various probability distributions. The desugaring makes the probability measure $\boldsymbol{y} := p(\boldsymbol{x})$ a rewriting condition, $p(\boldsymbol{x}) \longrightarrow \boldsymbol{y}$, and goes into a plain conditional rewrite rule. The probability distributions are implemented using special features in Maude for generating random numbers according to a *uniform* distribution (see [5, §9.3]). The uniform distribution is then used to sample into other distributions, for example

```
rl BERNOULLI(R) => if rand / rand-max < R then true else false fi .
```

is used to sample a Bernoulli distribution with success probability R, given as a rational number. The operator `rand` is treated specially by Maude; it returns a natural number between 0 and the constant `rand-max`.

As we noted above, both Real-Time Maude and PMaude are implemented as theory transformations. So via Real-Time Maude we can go from a real-time rewrite theory to a plain rewrite theory, and via PMaude we can go from a probabilistic rewrite theory to a plain rewrite theory. To combine the two paradigms what we have done is to manually apply the desugaring described above for PMaude within our larger real-time rewrite theory.

For analyzing probabilistic rewrite theories, it is often desirable to state logical queries *quantitatively*, that is, not with a *true* or *false* answer, but with a real number corresponding, for example, to a probability or, more generally, to some quantitative measurement of our system. For this reason, we use the QuaTEx probabilistic temporal logic of *Quantitative Temporal Expressions* proposed in [3]. This language is supported by the VeStA tool [26], which has an interface to Maude. The key idea of QuaTEx is to generalize probabilistic temporal logic formulas from Boolean-valued expressions to real-valued expressions. The Boolean interpretation is preserved as a special case using the real numbers 0 and 1. As usual, QuaTEx has *state expressions*, evaluated on states, and (real-valued) *path-expressions* evaluated on computation paths. The notion of state predicates is now generalized to that of *state functions*, which can evaluate quantitative properties of a state. QuaTEx is particularly expressive because of the possibility of defining recursive expressions. In this way, only the next operator (`#` in VeStA syntax) and conditional branching (`if` *Bexp* `then` *Pexp* `else` *Pexp*′ `fi`, with *Bexp* a Boolean expression and *Pexp*, *Pexp*′ path expressions) are needed to define more complex operators, such as "until". Model checking queries are given as expected values of path expressions. We refer to [3] for a detailed account of QuaTEx and its semantics. The VeStA tool [26] then performs *statistical model checking* on a probabilistic system by evaluating a QuaTEx expression on computation paths obtained by Monte Carlo simulation. The model checking query is parameterized by two values, α and δ, which are user-provided. VeStA responds to a query with a $(1 - \alpha) \cdot 100\%$ confidence interval bounded by δ for the expected value of the random variable defined by the QuaTEx formula.

Depending on the slack allowed by the given parameters, VeStA may need greater or fewer sample runs to compute this interval.

Using VeStA requires that the PMaude model be free from unrestrained nondeterminism that does not come from sampling a probability distribution. In particular, for the tool to work correctly it should never be the case that two rules apply to the same term. One way of solving this is described in [3]. There, a uniform distribution is used to give each event a (probabilistically) unique identifier and an event queue then orders the events by identifier. Each rewrite rule can only fire if the event associated with it is the one indicated by the front of the event queue. In our case this matching is done by adding a third field to the directed message construct and adding a special object with the event queue

```
op directed-msg : Receiver Msg EventID -> DirectedMsg .
class Control |
   event-queue : EventQueue .
```

Finally, we change each of the rules to match the directed message being consumed by the rule (all of our rules are of this form) with the the event at the front of the queue. We found that the details of handling events and the event queue are tricky to do cleanly. We did not want the intent of the rules to be obscured by a mass of syntax that simply deals with the event queue mechanism. The details of our eventual solution can be found in [11,1].

4.2 Refinement 1: Unsynchronized Timers

Our first refinement changes the round timers so that they are no longer globally synchronized. Instead, the timers are initially set to expire somewhere in the time interval $[0s, 10s]$, rather than at time 0 as in the idealized case. This is accomplished by adding a new message type , `init-msg`, and an associated *probabilistic rewrite rule*. In the starting configuration the only messages are the initialization messages, one for each sensor node, and we execute the rule below to get to the desired starting state where the protocol can be applied. The rule draws a value uniformly from $[0, 10]$ and uses this value to initialize the node's round timer. Note that we use a simplified syntax below, but the intent should be clear. In addition, we use (`...`) to indicate omitted code.

```
rl [init] : init-msg ... => delayed-msg(round-timer-msg, y) ...
      with probability y = uniform-dist(0(s), 10(s)) .
```

This rule is also used to initialize the node's location (not shown), therefore overcoming one of the problems with the analysis done in Section 3, namely, the inability to have the model checking analysis directly consider all (probabilistically) possible starting positions for the nodes.

Although this is a very simple change, it exposes a bug in the protocol. To see this, we first recast the connectedness property as the QuaTEx formula below. The key idea is to calculate the percentage of an interval, [`lower`, `upper`], when the network is totally connected. Whenever the tick rule is applied `PrctCon` looks

at the interval stepped over (using `#`, QuaTEx's *next* operator), determines if the network was connected, determines if the time interval overlaps [`lower`, `upper`], and concisely records this information.

```
PrctCon(x:Bool, y:Float, z:Float) =
   if time > lower and y < lower then
      #PrctCon(connected, time, z + Start(x, y));
   else if y >= lower and time <= upper then
      #PrctCon(connected, time, z + Mid(x, y))
   else if y > upper then
      100.0 * (z + End(x, y) / (upper - lower));
   else
      #PrctCon(connected, time, z);
   fi; fi; fi;
```

The `x` argument of `PrctCon` records whether or not during the previous time step the network was connected, `y` records the starting time of the previous time step, and `z` enumerates the time that the network is connected in the interval. The state expressions, such as `connected` and `time`, have the obvious meanings. The value returned by model checking the above QuaTEx formula in VeStA is an interval describing, statistically, the expected value of the *percentage of time the network was totally connected during the time interval* [`lower`, `upper`].

In Figure 1b we have plotted the expected value of `PrctCon` for the first ten global rounds of network operation. The result shows endemic disconnectedness during the first two global rounds. For comparison, we plot the same value for a probabilistic version of the idealized model in Figure 1a. In Section 5 we use a novel redesign methodology to fix the bug causing the disconnectedness.

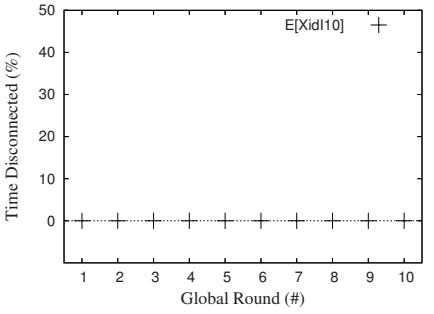
4.3 Refinement 2: Delay Uncertainty

The LMST protocol is highly localized in the sense that very little collaboration between nodes takes place. Therefore it is especially important to investigate the inter-node ordering of events and messages. Our second refinement addresses: (1) wireless transmission delays, (2) imperfections in quartz clock timers, and (3) the 802.11 MAC contention procedure [4]. Item (1), wireless transmission delay, is easily implemented using the `delayed-msg` construct defined in Section 3.1.

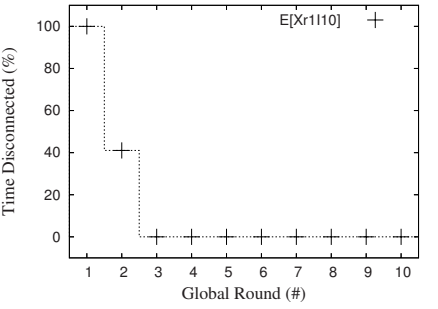
For realistic quartz clocks, additional infrastructure is needed. First, the wireless sensor node state gets a new attribute, `clock-drift`, and the initialization rule is modified so that each sensor node gets a value, uniformly distributed in the $[-5.0, 5.0]$ interval, for its clock drift. A slow clock is represented with negative values, and a fast clock with positive values. The absolute value gives the drift in *parts per million*.

In addition, we introduce a *Timer* sort, which we treat as a new type of *Delay*, defined by constructors

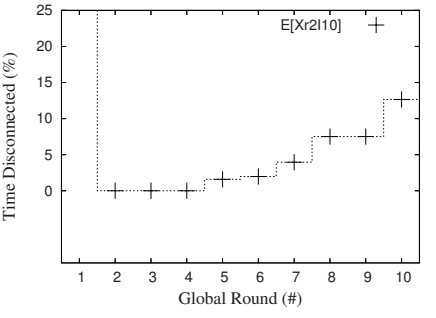
```
sort Timer  .  subsort Timer < Delay   .
op ACTIVE   : Float  -> Timer .
op SUS      : Float  -> Timer .
```



(a) Connectedness in idealized model.



(b) Connectedness in refinement 1.



(c) Connectedness in refinement 2.

Fig. 1. Quantified connectedness of the idealized model and refinements 1 and 2

The floating point arguments indicate the amount of time remaining according to the node's internal clock, which may be fast or slow. The two constructs differentiate between an active timer and one that has been suspended. Suspended timers are used as part of the 802.11 MAC contention protocol. Quartz clock drift is then modeled by replacing all event delays associated with timers with these constructs; plus a few auxiliary operations to update the constructs to account for drift. The main change that we need to make is in the definition of `delta` for the tick rule, which now must scale time decrements by clock drift. To do this we add a new equation to the definition of `delta`, exactly like the one in Section 3.1 but with

```
... delayed-msg(DMx, ACTIVE(Fx1 monus Tx with-scale Fx2))) ...
eq Fx monus Tx with-scale Fy =
      if Fy < 0.0 then
         Fx + float(Tx) - (float(Tx) * ((- Fy) / 1000000.0))
      else
         Fx + float(Tx) + (float(Tx) * ((Fy) / 1000000.0)) fi .
```

substituted for delayed messages having a regular delay. Note that we omit matching the drift (`Fx2`) in appropriate sensor node to save space. For suspended

timers, the `delta` function leaves the timer unchanged. There are some complications to deal concurrently with the event queue (see [11,1]).

The 802.11 MAC contention procedure defines how sensor nodes try to arbitrate access for the transmission medium. Too much noise in the medium can cause messages to be dropped, so the basic idea of the protocol is that each node measures the noise level and waits when it is too noisy. To model the 802.11 MAC contention procedure we define four new events:

```
msg difs-timer-msg      : -> Msg .
msg backoff-timer-msg   : -> Msg .
msg medium-busy-msg     : -> Msg .
msg medium-clear-msg    : -> Msg .
```

and a modification of the round timer rule. Under the 802.11 regime [4] when a sensor node wants to transmit a message (e.g. a hello message) it sets two timers. The DIFS timer is set for a fixed time period and is made active rightaway. The backoff timer is set *probabilistically* and waits for the DIFS period to finish before it becomes active. Therefore we change the round timer message event to emit two delayed messages according to the protocol (again, simplified syntax below)

```
rl [round-timer-msg] : round-timer-msg ... =>
      delayed-msg(difs-timer-msg    , ACTIVE(128(mu-s)))
      delayed-msg(backoff-timer-msg , SUS(y * 50(mu-s)) ...
         with probability  y := uniform-dist(0, 15) .
```

The interaction between the two timers is governed by the *carrier sense mechanism*, which is implemented using the other two new message types defined above. In addition to these messages we add a new field to our sensor node class called `medium-busy`, which takes a natural number value and records when the medium is busy. The value is 0 when the medium is not busy, gets incremented on every `medium-busy-msg`, and decremented on every `medium-clear-msg`.

Due to space limitations we cannot give all of the details of our 802.11 model (see [11] for full details). The part of the protocol that we have not specified here defines how the two timers respond to changes in the carrier sense mechanism. The basic idea is that whenever the medium becomes busy (`medium-busy` field goes from 0 to 1), the backoff timer is suspended and the DIFS timer gets reset. When the backoff timer finishes, then the node can finally transmit its message. Since we are only concerned with hello messages, whenever a backoff timer message is consumed, the node transmits its hello message.

Analysis results are plotted in Figure 1c and show real disconnectedness during portions of the first ten rounds. The significance of such a level of disconnectedness depends on many variables. What we found though is that an expected disconnectedness value of $x\%$ usually corresponds to an $x\%$ chance that the network is disconnected for the entire round. The bugs causing the disconnectedness are identified and fixed in Section 5.

5 A New Realistic Design of LMST

The methods of statistical quantitative analysis have so far been used to diagnose serious issues with the LMST protocol, but, unfortunately, the problems

uncovered remain otherwise idiopathic. Determining the *causes* of the undesired behavior is a crucial part of the formal analysis process. In this section we propose a new method for redesigning probabilistic systems by statistical quantitative analysis. Instead of calculating expected values $E[X]$ directly, we calculate *mathematical correlations*, $\rho(X, Y)$, of two random variables. The idea is that one of the random variables represents the symptoms of a bug, and the other a hypothesized cause. This method can save significant time during the debugging and redesign process by providing a way to isolate the cause of the a bug *without first implementing a hypothesized fix*, which might require substantial effort, and only afterward trying again to detect the bug in the "fixed" model.

5.1 A Formally-Based System Redesign Methodology

Debugging and redesigning a complex protocol is rarely a trivial task, particularly when the protocol is concurrent and/or probabilistic, because specific symptoms are usually difficult to reproduce. For concurrent finite state systems one can witness bugs directly using standard model checking algorithms to generate counter-examples. These can then be used to diagnose the cause of a bug and guide the effort required to actually fix it. However, for probabilistic systems and approximate quantitative analysis no analogue seems to exist.

Moreover, the process of determining the cause of a bug is extremely important, because otherwise, without a known cause, any effort used to modify the model can easily be wasted if the changes do not happen to hit on the cause. This wasted effort can be substantial if, for example, a fix does not fit cleanly within the current architecture of the model specification. Therefore it is essential to know beforehand that such efforts will likely result in success.

Our proposed methodology to address these issues begins with the idea of using statistical quantitative analysis to calculate *mathematical correlations*. The correlation of two random variables, X representing the *symptoms*, and Y the *hypothesized cause*, is calculated to establish the likelihood of a causal relationship between X and Y. Of course, X and Y being correlated does not *prove* a causal relationship, but it is a necessary condition for causality and indicative of it. Since specifying X and Y, for example as QuaTEx formulas, usually requires much less effort than directly modifying the model, the process can lead to reduced debugging and redesign time. However, unlike the finite state concurrent system case, some extra work must be done to formulate Y. The proposed system redesign methodology assumes two initial items:

- A probabilistic model $\mathcal{M}$, for example a PMaude module.
- An observable value measuring, for any given sample from the probability space defined by $\mathcal{M}$, fit or disagreement with required behavior. This takes the form a random variable X over the sample probability space and can be defined by a QuaTEx formula.

The methodology then proceeds through the following steps:

1. Use statistical quantitative analysis, using tools such as by VeStA or PRISM [14], to calculate $E[X]$. If the value indicates buggy behavior, then go on with the remaining steps, otherwise there is no need to go further.
2. Hypothesize a cause for the bug. The hypothesized cause can be formulated also as a QuaTEx formula, yielding a second random variable, Y, on the sample probability space, $\mathcal{M}$.
3. The next step is to calculate the *correlation coefficient* (see [25]), $\rho(X, Y)$, which is easily accomplished by observing the following expansion, which turns $\rho(X, Y)$ into a simple expression of expected values

$$\rho(X,Y) = \frac{\operatorname{Cov} XY}{\sigma(X)\sigma(Y)} = \frac{E[XY] - E[X]E[Y]}{\sqrt{E[(X - E[X])^2]}\sqrt{E[(Y - E[Y])^2]}}$$

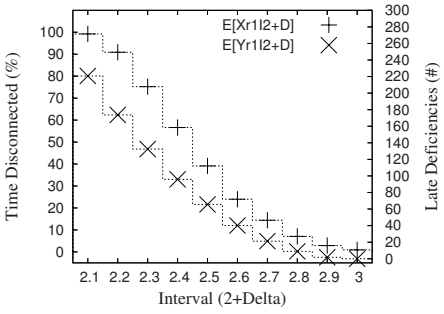
   Therefore, statistical quantitative analysis is used to first calculate $E[X]$, $E[Y]$, and $E[XY]$; and then secondarily using these values we calculate $E[(X - E[X])^2]$ and $E[(Y - E[Y])^2]$. The final value of $\rho(X, Y)$ is easily calculated from these values. If the correlation is not significant (i.e. close to 0) then this step is repeated with a new hypothesis, otherwise we go on.
4. Modify the model $\mathcal{M}$ to remove the cause articulated through Y, yielding a new model $\mathcal{M}'$. For $\mathcal{M}'$ calculate, via statistical quantitative analysis, the expected value of X', which captures the same observable value as X but is a random variable over the probability space defined by $\mathcal{M}'$ instead of $\mathcal{M}$. If $E[X']$ shows no buggy behavior then we finish, otherwise repeat.

We feel that this method adds rigor to the process of redesigning a probabilistic system, and does so without a lot of auxiliary, extraneous work that would otherwise be needed. Compared to a more informal method, the extra burden on the designer is only in expressing Y concretely. While certainly not always trivial, writing down the hypothesized cause has a number of benefits outside of just fixing the immediate bug. For example, it provides concrete documentation of the bug, assurances that the bug has truly been fixed, and an easy regression test to make sure the bug is not re-introduced in the future. In Section 5.2 we apply this methodology to determine the causes of the bugs uncovered in Section 4. Using this knowledge we then redesign the LMST protocol to reach a realistic, implementable design which ensures connectedness.

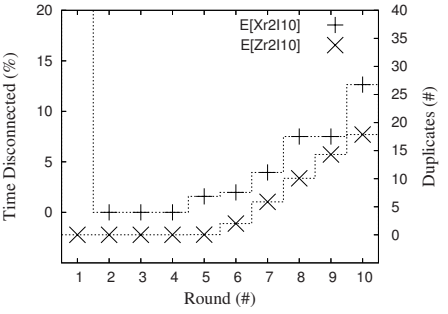
5.2 Redesign of the LMST Protocol

Let us now consider applying the redesign methodology to our model of the LMST protocol. The analysis results for refinements 1 and 2 (Figure 1) show significant disconnectedness behavior. With the first refinement the disconnectedness appears to be transient, but the second refinement results in pervasive disconnectedness across global rounds. We fix both bugs in this section.

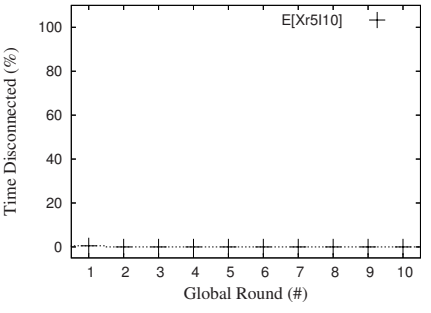
According to the methodology outlined in the previous section, we need to define a random variable X describing the correctness property that we are



(a) Debugging of refinement 1.



(b) Debugging of refinement 2.



(c) Connectedness in our LMST redesign.

Fig. 2. Causal analyses used in our LMST redesign

interested in. This poses a subtle problem, given the data presented in Figure 1, because we evaluated *multiple separate* QuaTEx formulas, one for each global round, to create the plot. Our solution to making this set of random variables into a single one is to, for each simulation run, encode a probabilistically chosen global round to evaluate the disconnectedness property on.

We first look at refinement 1, and the set of intervals partitioning global round 2 into 10 parts. We still need to hypothesize and quantify a *cause* for the undesired behavior witnessed through X. The hypothesis that we made was that the bug involved something we will call *late neighbors*. Late neighbors are visible neighbors that do not make themselves known to other nodes within their transmission range during global round 1 because their local round *begins later*. Since they send their hello messages late, they are not received in time to be incorporated into the (early) neighbor's first local topology update. Due to space limitations, we again refer to [11,1] for full details.

Figure 2a plots connectedness versus the total number of late neighbors for nodes that have not yet begun their second local round. It is clear that the two plots exhibit strong correlation, which can be calculated for the random variables just described using the methods of the previous section. The resulting

correlation is 0.99, thus indicating that we should make sure that the problem of late neighbors is fixed.

The situation with refinement 2 is even more interesting, because, as seen in Figure 1c and re-printed in Figure 2b, the disconnectedness behavior of this model is persistent across rounds. We conjectured that what might be happening was something like the following: one node with a fast clock processes a round timer event so early, relative to a neighbor, that it sends its round $i + 1$ hello message before the second node processes its round i message. Therefore, the second node gets a duplicate message from the first node.

To precisely quantify this conjecture, we defined a QuaTEx formula to calculate the number of duplicate messages where the node sending the duplicate message has a clock that is fast relative to its neighbor. Figure 2b plots results for both random variables, and it is easy to see in this graph that the two are correlated. The exact value is -0.13, indicating significant correlation, but not as high as we would like for such a small number of intervals. However, with the outlier for global round 1 removed the correlation rises to 0.98. Therefore, it seems reasonable to try to fix the bug by removing duplicate messages.

We want to fix both bugs uncovered above: the one due to late neighbors and the other due to duplicated messages from clock drift. The late neighbors problem is easily solved by *each node broadcasting a hello message as soon as it turns on*. To prevent clock drift and MAC contention from causing duplicate messages, we take a two-pronged approach. Since drift and contention only cause small perturbations in the amount of time between successive hello messages, our solution is to *delay topology updates for a few milliseconds after a round timer event*. However, this does not completely solve the problem and in fact only delays it as the clocks drift farther apart. So the second component of our fix is to *apply an ultra-lightweight clock synchronization algorithm [16]*. The overhead of the algorithm is that now each hello message gets time-stamped.

Our new design, with the three fixes explained above, completely removes the problems that we observed in refinements 1 and 2. Correctness results, based on quantitative statistical analysis of our new design, are given in Figure 2c.

6 Related Work and Conclusions

Alternatives to Real-Time Maude [22] include Uppaal [17] and HyTech [7], which use timed and linear hybrid automata as the underlying modeling formalisms. Compared with Real-Time Maude, Uppaal and HyTech trade expressibility in the modeling language for certain decidability results.

There are various alternatives to PMaude and VeStA. The PRISM tool [14] supports a BDD-based probabilistic model checking algorithm and also an approximate, statistical model checking analysis through Monte Carlo techniques. PRISM supports a large number of modeling formalisms [14]. In [12] the algorithm used by VeStA is refined so that an the sample size necessary to achieve normality in the data can be computed before analysis begins. The modeling language used is probabilistic rewrite theories. Other probabilistic model checking tools include APMC

[8], $E \vdash MC^2$ [9], and Rapture [10]. As with Real-Time Maude, the most significant difference between PMaude/VeStA and other tools is the tradeoff between expressiveness and algorithmic power.

The combination of PMaude and VeStA has been used in multiple case studies, including the analysis of a DoS resistant TCP/IP protocol [2] and two case studies involving distributed object-based stochastic hybrid systems [20]. In addition there are other case studies on 802.11 [15] and sensor networks [6].

In conclusion, this work has presented two main contributions. First, we have extended the formal specification and analysis approach for wireless sensor networks advocated by Ölveczky and Thorvaldsen [23,24] to the probabilistic and statistical model checking setting, demonstrating significant flaws in a realistic protocol. Second, we have presented a system redesign methodology applicable to probabilistic systems in general and wireless sensor networks in particular where QuaTEx-based quantitative measurement of bugs and their hypothesized causes can be correlated; and the knowledge thus gained can be used to redesign a system free of the given bugs.

Acknowledgment

This research was supported by The Boeing Company, Grant C8088-557395.

References

1. `http://peepal.cs.uiuc.edu/~katelman/fmoods_2008.tgz`
2. Agha, G., Gunter, C., Greenwald, M., Khanna, S., Meseguer, J., Sen, K., Thati, P.: Formal Modeling and Analysis of DoS Using Probabilistic Rewrite Theories. In: Foundations of Computer Security (FCS) (2005)
3. Agha, G., Meseguer, J., Sen, K.: PMaude: Rewrite-based Specification Language for Probabilistic Object Systems. In: Proc. of the Third Workshop on Quantitative Aspects of Programming Languages (QAPL 2005) (2005)
4. ANSI/IEEE. ANSI/IEEE Std 802.11, Edition (R2003) (1999)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. Springer, Heidelberg (2007)
6. Demaille, A., Hérault, T., Peyronnet, S.: Probabilistic Verification of Sensor Networks. In: Intl. Conf. on Research, Innovation and Vision for the Future (2006)
7. Henzinger, T.A., Ho, P.-H., Wong-Toi, H.: HyTech: A Model Checker for Hybrid Systems. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 460–463. Springer, Heidelberg (1997)
8. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate Probabilistic Model Checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937. Springer, Heidelberg (2004)
9. Hermanns, H., Katoen, J.-P., Meyer-Kayser, J., Siegle, M.: A Markov Chain Model Checker. In: Schwartzbach, M.I., Graf, S. (eds.) ETAPS 2000 and TACAS 2000. LNCS, vol. 1785. Springer, Heidelberg (2000)
10. Jeannet, B., D'Argenio, P.R., Larsen, K.G.: Rapture: A tool for verifying Markov Decision Processes. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421. Springer, Heidelberg (2002)

11. Katelman, M., Meseguer, J., Hou, J.: Formal Modeling, Analysis, and Debugging of a Localized Topology Control Protocol with Real-Time Maude and Probabilistic Model Checking. Technical report, University of Illinois at Urbana-Champaign (in preparation, 2008)
12. Kim, M., Stehr, M.-O., Talcott, C., Dutt, N., Venkatasubramanian, N.: A Probabilistic Formal Analysis Approach to Cross Layer Optimization in Distributed Embedded Systems. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468. Springer, Heidelberg (2007)
13. Kumar, N., Sen, K., Meseguer, J., Agha, G.: A Rewriting Based Model for Probabilistic Distributed Object Systems. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884. Springer, Heidelberg (2003)
14. Kwiatkowska, M., Norman, G., Parker, D.: Quantitative analysis with the probabilistic model checker PRISM. ENTCS 153(2), 5–31 (2005)
15. Kwiatkowska, M., Norman, G., Sproston, J.: Probabilistic Model Checking of the IEEE 802.11 Wireless Local Area Network Protocol. In: Proc. of the Second Joint Intl. Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM-PROBMIV 2002) (2002)
16. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM 21(7) (1978)
17. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Intl. Journal on Software Tools for Technology Transfer 1(1-2), 134–152 (1997)
18. Li, N., Hou, J.C., Sha, L.: Design and Analysis of an MST-Based Topology Control Algorithm. In: INFOCOM 2003 (2003)
19. Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. Theor. Comput. Sci. 96(1), 73–155 (1992)
20. Meseguer, J., Sharykin, R.: Specification and Analysis of Distributed Object-Based Stochastic Hybrid Systems. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 460–475. Springer, Heidelberg (2006)
21. Ölveczky, P.C., Meseguer, J.: Specification of Real-Time and Hybrid Systems in Rewriting Logic. Theoretical Computer Science 285, 359–405 (2002)
22. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher Order and Symbolic Computation 20(1-2) (2007)
23. Ölveczky, P.C., Thorvaldsen, S.: Formal modeling and analysis of wireless sensor network algorithms in Real-Time Maude. In: 20th Intl. Parallel and Distributed Processing Symposium (IPDPS 2006) (2006)
24. Ölveczky, P.C., Thorvaldsen, S.: Formal Modeling and Analysis of the OGDC Wireless Sensor Network Algorithm in Real-Time Maude. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468. Springer, Heidelberg (2007)
25. Parzen, E.: Modern Probability Theory and Its Applications. John Wiley & Sons, Inc., Chichester (1960)
26. Sen, K., Viswanathan, M., Agha, G.: VESTA: A Statistical Model Checker and Analyzer for Probabilistic Systems. In: 2nd Intl. Conf. on the Quantitative Evaluation of Systems (2005)

A Minimal Set of Refactoring Rules for Object-Z

Tim McComb[1] and Graeme Smith[2]

[1] ARC Centre of Excellence in Bioinformatics
Institute for Molecular Bioscience, The University of Queensland, Australia
[2] School of Information Technology and Electrical Engineering
The University of Queensland, Australia

Abstract. This paper presents a minimal and complete set of structural refactoring rules for the Object-Z specification language that allow for the derivation of arbitrary object-oriented architectures. The rules are equivalence preserving and work in concert with existing class refinement theory, so that any design derived using the rule set can be shown to be equivalent to, or a refinement of, the original specification.

1 Introduction

Class instantiation, class inheritance, polymorphism, and generics (class parameters or templates) are four object-oriented architectural constructs which are almost universal. They underpin the paradigm and provide the modularity and reuse capabilities. Object-oriented formal specification languages such as Object-Z [14], Alloy [6], and VDM^{++} [7] share these core features with their programming language counterparts. However, the way they are utilised to capture requirements associated with a problem domain is often quite different from the way in which they are used to implement a specific solution to a problem. The result is that an object-oriented specification does not usually directly resemble, in a structural sense, the design of the desired implementation. Here, structure is interpreted as the relationships between classes. Generally, a set of specification classes will describe a system of many more interacting implementation classes.

To bridge this gap between specification and implementation, specification refactoring rules have been proposed [7,8,9,11,2,5]. These allow the structure of a specification to be incrementally transformed to represent a given design. Goldsack and Lano [7,8], for example, introduced the Annealing rule to VDM^{++}. This rule effectively splits a class's state and operations into two classes — one holding a reference to an instance of the other. It was later adapted to Object-Z by McComb [9] who also introduced the Coalescence rule. This second rule merges two classes together to create a new class that simulates both. Together these rules have been shown quite effective for introducing designs [11].

However, these rules both deal with referential structure, and do not cover the other primary forms of object-oriented design structure: inheritance, polymorphism and generics. In this paper, we fill this gap by formalising rules for

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 170–184, 2008.

Object-Z that permit the modification of inheritance hierarchies and allow classes to be parameterised. Furthermore, we show that our set of rules is both minimal and complete for refactoring Object-Z specifications to derive designs.

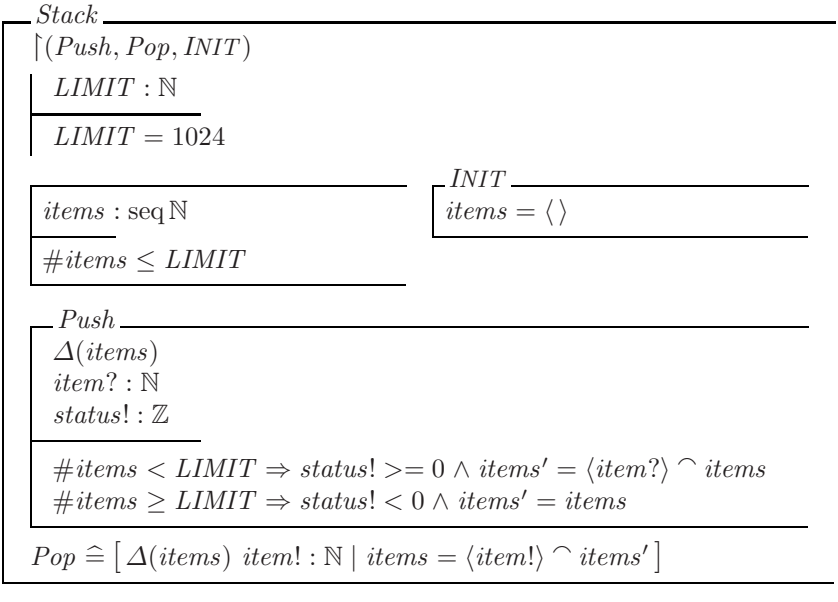
We begin in Section 2 with an overview of the Object-Z language. We then provide rules to introduce generic class parameters, to introduce polymorphic behaviour, and to introduce inheritance in Sections 3 to 5 respectively. Together with the ANNEALING rule, these three rules have been shown to be *complete* in the sense that any reasonable design can be derived from any specification [10]. The proof is outlined in Section 6 before we conclude in Section 7.

2 Object-Z

Object-Z [14] extends the formal specification language Z [16] with explicit support for the fundamental constructs of object orientation: (generic) classes, objects, inheritance and polymorphism. Here we overview the notation for basic classes and objects. Other notation will be introduced in the following sections.

A class in Object-Z groups together a collection of state variables with their initial conditions and a set of operations which may change their values. The state variables, initialisation predicate, and operations of a class are collectively referred to as its *features*. The interface of the class is specified by a *visibility list* of the form $\upharpoonright(\ldots)$ listing its externally accessible features.

Consider, for example, the following Object-Z class *Stack*.

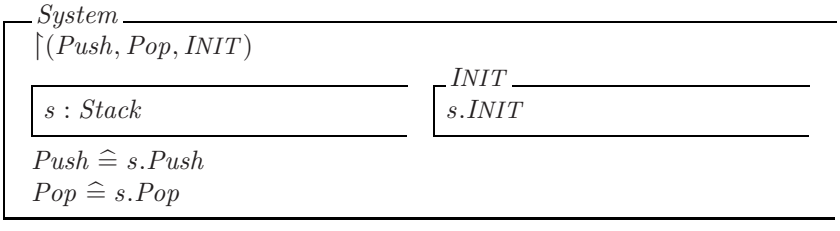


The class *Stack* has a state variable *items* of type $\mathrm{seq}\,\mathbb{N}$ (a sequence where the elements are natural numbers). The possible bindings of values for state variables are constrained initially by the initialisation predicate, and by the invariant predicate contained within the state schema. Hence, the *items* sequence is initially empty.

Each operation is a schema describing the relationship between pre- and post-state variables. A variable decorated with a prime, e.g., x', denotes the post-state value. All post-state variables from the state schema are available to operations, but by default they are equated to their pre-state counterparts (they do not change). Operations may introduce constraints over post-state variables by including them in a *delta-list* of the form $\Delta(\ldots)$. Any variables included in the delta-list have their pre-/post-state equality constraint relaxed.

For example, the operation *Push* from the *Stack* class concatenates ($\frown$) the input *item*? to the beginning of the *items* sequence, if the size of *items* is less than the *LIMIT*. Thus, *items* appears in the delta-list of *Push*. An output variable *status*! is set to be greater-than or equal-to zero if *items* can indeed accommodate the new item, otherwise the *status*! binds to a value strictly less-than zero and *items* remains unchanged. The operation *Pop* in this example removes the first item, bound to output variable *item*!, from the beginning of the sequence.

Such a class can be instantiated within another class and its visible features accessed using standard dereferencing notation. For example, consider the following class *System* which references a single object of class *Stack*.



Given the declaration $s : Stack$, the notation $s.INIT$ denotes a predicate which is true precisely when the referenced object is in its initial state. Also, the notation $s.Push$ is an operation corresponding to the referenced object undergoing its *Push* operation, and $s.Pop$ is an operation corresponding to the referenced object undergoing its *Pop* operation.

3 Introduce Generic Parameter

This section provides a description of the Introduce Generic Parameter refactoring rule. Generic parameters in Object-Z allow a type, or a list of types, to be passed as parameters to a class. Refactoring a specification to add support

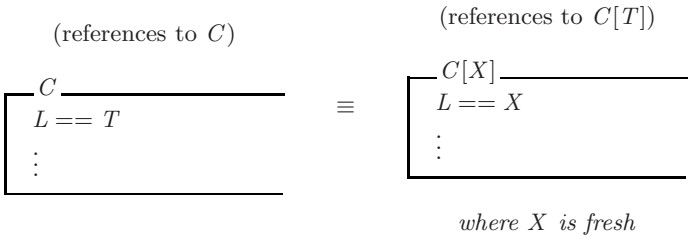


Fig. 1. Introduce generic parameter refactoring

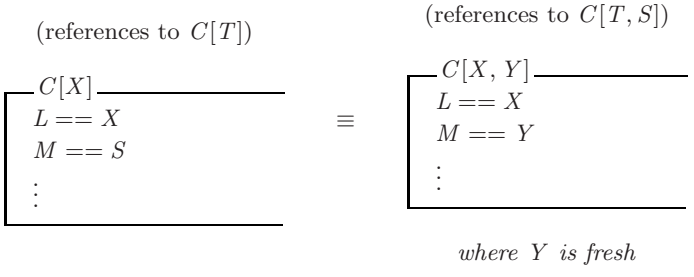


Fig. 2. Repeated application of the introduce generic parameter refactoring

for generic parameterisation of classes is desirable, as it allows for the derivation of library components, and increases reuse throughout the design as a single class may be instantiated many times with different parameters. The parameterised classes in Object-Z could possibly be implemented using the support for *generics* in Java [1] or *templates* in C^{++} [15].

The rule is illustrated in Figure 1. A class C has a locally defined type L which is defined to be the actual type T. When the rule is applied, the class C is replaced with a class $C[X]$, where the name X is fresh, and the local definition which previously defined L as T is changed to define L as X. All references to C in the specification are replaced with references to $C[T]$, including references for inheritance.

This refactoring rule only introduces one parameter, but repeated application can provide as many parameters as necessary. As new parameters are added, they can be appended to the right of the parameter list. For example, Figure 2 represents the result of the rule being applied again to the right-hand side of Figure 1, introducing a new parameter Y to stand for an actual type S. Exactly where in the list of parameters the new parameter is inserted is arbitrary, as long as the references are updated in a consistent manner.

The soundness of this rule follows directly from the semantics of generic parameters in Object-Z. A formal proof is presented in [10].

3.1 Example

The *Stack* in Section 2 specifically deals with natural numbers, but through the application of the above refactoring rule we are able to systematically introduce a generic parameter.

First, the *Stack* class must undergo a refinement step to introduce a local definition, such that the class conforms to the precondition required by the refactoring rule. The refinement step (actually, an equivalence transformation) would be proved using the simulation rules for Object-Z [3]. We present just the result of the refinement step here.

Stack

$\lceil(Push, Pop, INIT)$

$L == \mathbb{N}$

$LIMIT : \mathbb{N}$

$LIMIT = 1024$

$items : \operatorname{seq} L$

$\#items \leq LIMIT$

INIT

$items = \langle\,\rangle$

Push

$\Delta(items)$

$item? : L$

$status! : \mathbb{Z}$

$\#items < LIMIT \Rightarrow status! >= 0 \land items' = \langle item? \rangle \frown items$

$\#items \geq LIMIT \Rightarrow status! < 0 \land items' = items$

$Pop \mathrel{\widehat=} [\, \Delta(items)\ item! : L \mid items = \langle item! \rangle \frown items' \,]$

We are now in a position to apply the refactoring rule, which introduces the parameter X to *Stack* and updates the reference to *Stack* in the *System* class to instantiate the parameter with $\mathbb{N}$.

System

$\lceil(Push, Pop, INIT)$

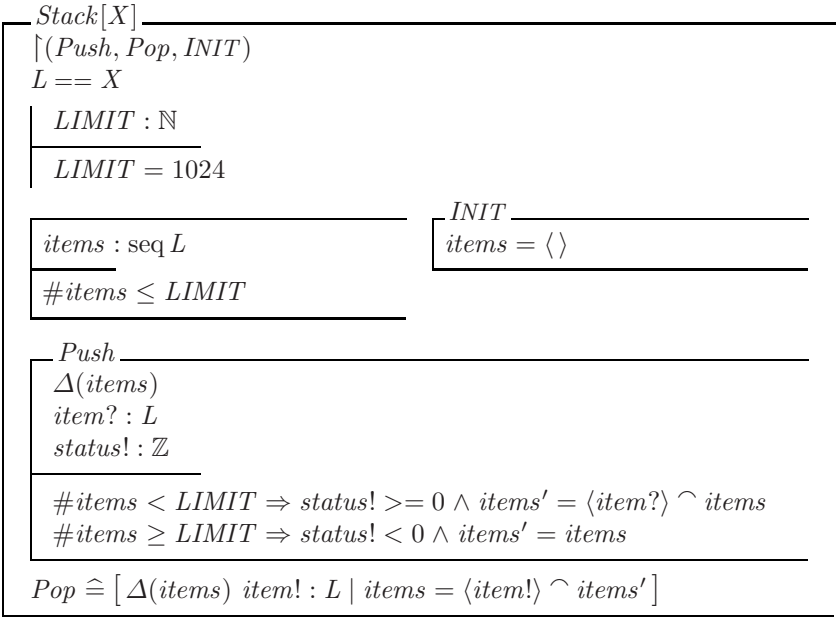
$s : Stack[\mathbb{N}]$

INIT

$s.INIT$

$Push \mathrel{\widehat=} s.Push$

$Pop \mathrel{\widehat=} s.Pop$



4 Introduce Polymorphism

Not all inheritance hierarchies take advantage of polymorphism in Object-Z, and conversely not all polymorphism must be confined to inheritance hierarchies. In programming languages like Java it is common to have classes implement *interfaces* that provide a mechanism for polymorphism that is not related to inheritance. This orthogonal treatment of polymorphism both in Object-Z and in some programming languages warrants a rule specifically for its introduction, rather than treating it as a by-product of introducing inheritance.

In Figure 3, the class $C[P_1, \ldots, P_m]$ on the left-hand side has exactly $n+1$ means of referencing it: by $C[P_1, \ldots, P_m]$, or by n axiomatically defined aliases $A_1[P_1, \ldots, P_m]$,...,$A_n[P_1, \ldots, P_m]$ which disjointly partition the references to objects of $C[P_1, \ldots, P_m]$. The parameter lists $[P_1, \ldots, P_m]$ are irrelevant to the application of the rule, except that all classes $A_1, \ldots, A_n$ and C must have the same arity. For brevity of presentation these parameter lists are omitted from the discussion below.

The introduction of polymorphism is normally motivated by the identification of a class (C) that behaves in different ways depending upon the context in which it is used. The Introduce Polymorphism rule requires that the designer identify the contexts where alternate behaviours are expected, and divide the references between $A_1, \ldots, A_n$ accordingly. Since the collection of class aliases $A_1, \ldots, A_n$ are together disjoint, any introduced references to these classes must be proved to be invariantly unequal whenever the references are to different aliases. This can be proved as a data refinement.

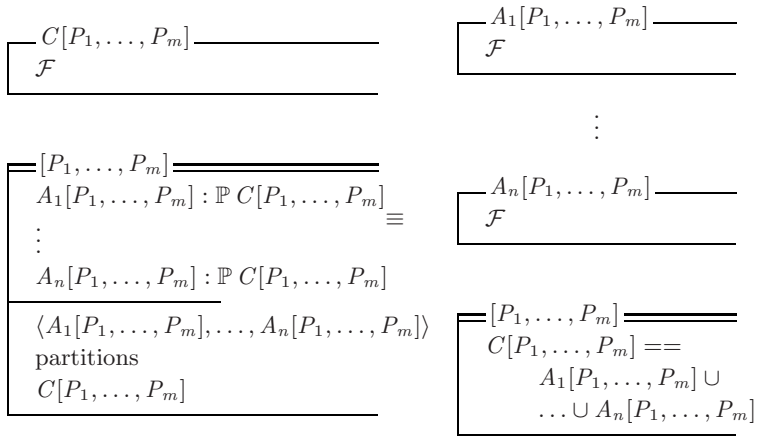


Fig. 3. Introduce polymorphism refactoring

Assuming this identification and partitioning of object references has occurred, the INTRODUCE POLYMORPHISM rule allows for the splitting of the behaviours into separate class definitions (A_1 to A_n on the right-hand side of Figure 3). To execute the refactoring transformation, all of the features of class C are copied verbatim to define the classes A_1 to A_n. The class C is removed from the specification, but C is globally defined to be the class union $A_1 \cup \ldots \cup A_n$ — thus providing for the polymorphism. The identical feature sets of the classes are represented with the symbol $\mathcal{F}$ in Figure 3.

Since C is defined as a class union after the application of the transformation, C cannot be inherited by any other classes in the specification after this refactoring is applied (this is a restriction of the Object-Z language [14]). Such classes must inherit one of $A_1, \ldots, A_n$ instead.

There is an axiomatic definition on the left-hand side that describes the typing relationships between $A_1, \ldots, A_n$ and C. The designer must add this to the specification as a precondition to applying the rule. The axiomatic definition is not only important to declare the meaning of $A_1, \ldots, A_n$ (i.e., that they are aliases for class C) but if the rule is applied in reverse (to *coalesce* two or more classes), this axiomatic definition retains the vital information that relates the types.

There does not need to be any distinction between the behaviours of the aliases $A_1, \ldots, A_n$, but without it this would render the application of the rule largely redundant. When there is a distinction, it is expected that the different behaviours are explicitly guarded. For example, the designer may wish for an operation Op in C to behave in two different ways captured by the operation schemas α and β, depending upon its context. The identification of these contexts is achieved by instantiating C as A_1 or A_2 respectively. The designer then guards these behaviours through the use of the choice operator $[\!]$ and the *self* keyword inside the operation: using $Op \mathrel{\widehat=} ([self \in A_1] \wedge \alpha) \,[\!]\, ([self \in A_2] \wedge \beta)$, ensuring that the introduction of the guards does not affect the behavioural interpretation

of the specification (e.g., whenever a reference to A_1 is introduced, α would have always been the behaviour of *Op* prior to the application of the rule).

Although the class definition is copied, the use of these guards becomes crucial after the application of the refactoring. The designer can substantially simplify the class definitions $A_1, \ldots, A_n$ by realising that in class A_1 (from the example above), $[self \in A_1] \equiv [\text{true}]$ and $[self \in A_2] \equiv [\text{false}]$; and likewise in class A_2, $[self \in A_2] \equiv [\text{true}]$ and $[self \in A_1] \equiv [\text{false}]$. Therefore, in class A_1, *Op* simplifies to α. Similarly, in class A_2, *Op* simplifies to β.

It is particularly important to realise that for instances $a : A_1;\ b : A_2$, it is never the case that $a = b$ because the references are disjoint ($A_1 \cap A_2 = \varnothing$). If A_1 and A_2 objects need to reside in a common data structure, for example a set declaration using the class union $A_1 \cup A_2$, then references to C may be used.

The labels $A_1, \ldots, A_n$ and C are representative only: the rule requires that $A_1, \ldots, A_n$ are fresh names, and generic parameters are carried across ($C[X]$, for example, becomes $A_1[X], \ldots, A_n[X]$). A proof of soundness of the rule is presented in [10].

4.1 Example

The Introduce Polymorphism rule tends to act as a bridge for applying more interesting refactorings: it is best utilised in combination with the other rules and interesting class refinements. So to provide an example for the Introduce Polymorphism refactoring, we are going to progress towards deriving a class *RepStack*$[X]$ which inherits *Stack*$[X]$. The class *RepStack*$[X]$ will define a *reporting* stack, where more helpful information is provided in the *status*! output of the *Push* operation. This requires inheritance, for which a rule will be provided in the Section 5, so we will only make progress toward the final goal at this stage (the example will be completed at the end of the Section 5).

We begin our example where the Introduce Generic Parameter rule ended, with the *System* class and a parameterised *Stack*$[X]$ class (refer to Section 3.1). In order to satisfy the prerequisite for the rule (being applied in the forward direction), we must introduce some new (generic) class aliases into the specification. As the new identifiers are fresh, this cannot affect the meaning of the specification.

$$
\begin{array}{|l}
\hline
[X] \\
\hline
RegularStack[X], RepStack[X] : \mathbb{P}\, Stack[X] \\
\hline
\langle RegularStack[X], RepStack[X] \rangle \text{ partitions } Stack[X] \\
\hline
\end{array}
$$

Although we have now satisfied the precondition for applying the rule to the *Stack*$[X]$ class, now is a good time to perform a refinement upon the *System* class to reference *RepStack*$[X]$ rather than *Stack*$[X]$. This refinement is correct, as *RepStack*$[X]$ is an alias for *Stack*$[X]$ (and therefore has the same meaning), so we omit a detailed argument as to its correctness and just present the result of the refinement:

System

$\lceil (Push, Pop, INIT)$

$s : RepStack[\mathbb{N}]$

INIT

$s.INIT$

$Push \mathrel{\widehat=} s.Push$

$Pop \mathrel{\widehat=} s.Pop$

The reason we have chosen to perform the refinement at this stage is because it is easier to justify (as $RepStack[X]$ is behaviourally equivalent to $Stack[X]$), and we know that we wish to have *System* reference $RepStack[X]$ at the end of the refactoring process.

We now apply the Introduce Polymorphism rule to the $Stack[X]$ class. The yields two class paragraph definitions $RegularStack[X]$ and $RepStack[X]$ which exactly match (apart from the class name) the definition of $Stack[X]$ prior to the rule being applied. The definition of $Stack[X]$, in turn, becomes:

$[X]$

$Stack[X] == RegularStack[X] \cup RepStack[X]$

Our intention is to have $RepStack[X]$ inherit $RegularStack[X]$, which will become possible with the Introduce Inheritance refactoring rule presented in Section 5.

5 Introduce Inheritance

Reuse of data constructs and operations in classes is achieved through inheritance in the object-oriented paradigm (both with programming and specification). The Introduce Inheritance rule offers a means by which to build an inheritance hierarchy from existing classes.

The Introduce Inheritance rule creates an inheritance relationship between any two classes in the specification, as long as the addition of the relationship does

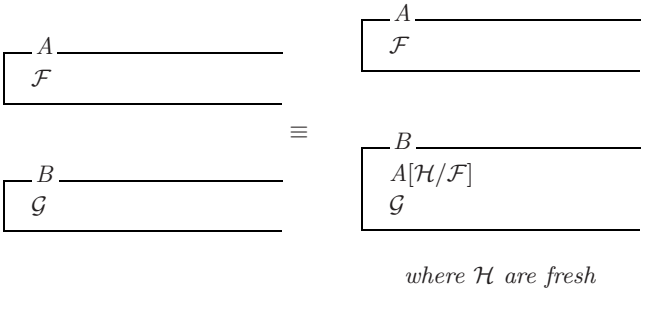


Fig. 4. Introduce inheritance refactoring

not result in a circular dependency. Figure 4 illustrates the application of the rule to two classes A and B with features $\mathcal{F}$ and $\mathcal{G}$ respectively.

The rule is most effectively applied to link together classes that contain common features in order to maximise the potential for reuse, but the classes need not share any features at all. This is because the INTRODUCE INHERITANCE rule not only adds the inheritance relationship (indicated in Figure 4 by the inclusion of A in B) but also hides every feature of the superclass by assigning them a fresh name (the notation '$\mathcal{H}/\mathcal{F}$' indicates that all features $\mathcal{F}$ of the superclass A are hidden by assigning fresh names $\mathcal{H}$ for the features). The combination of inheritance and hiding makes the refactoring rule an equivalence transformation, so long as we assume that the inheritance-based polymorphism operator $\downarrow$ is not used in the specification. $\downarrow C$ is an abbreviation for the class union of C with all of its subclasses. All occurences of $\downarrow$ in the specification must be replaced to specifically enumerate the inheritance hierarchy as a class union (that is, the $\downarrow$ must be replaced by its definition).

Some classes have an implicit visibility-list, whereby it is understood that every feature of the class is externally visible [14]. In such circumstances, the rule is still equivalence preserving as the interface of B is only widened, and the fresh features cannot possibly have been referenced as they did not previously exist.

To use the features inherited from the superclass, the designer must make refinements local to the subclass to reference the features in $\mathcal{H}$. Note that the introduced reference to the superclass must parameterise the superclass if it has generic parameters. These parameter instantiations of the superclass reference may include formal parameters of the subclass.

The reversal of the INTRODUCE INHERITANCE rule removes the inheritance relationship under the condition that every feature of the superclass concerned is not referenced ("fresh"). As inheritance in Object-Z is syntax-based [14], this precondition can be satisfied by copying any referenced feature definitions from the superclass into the subclass. A soundness proof for the rule is presented in [10].

5.1 Example

The example of the previous section (4.1) was left unfinished owing to the absence of the INTRODUCE INHERITANCE rule described above; we are now in a position to complete it.

The two classes, *RegularStack*$[X]$ and *RepStack*$[X]$, are identical. We wish to break this symmetry, however, by encapsulating the general stack behaviour in the *RegularStack*$[X]$ class and extending that behaviour in the *RepStack*$[X]$ class via inheritance. The refactoring rule can be applied immediately to the specification at the end of Section 4.1 to yield an altered definition for *RepStack*$[X]$:

RepStack[*X*]

$\lceil(Push, Pop, INIT)$

$RegularStack[X][SuperL/L, SuperLIMIT/LIMIT, SuperPush/Push, SuperPop/Pop, SuperItems/items, SuperInit/INIT]$

$L == X$

$LIMIT : \mathbb{N}$

$LIMIT = 1024$

$items : \operatorname{seq} L$

$\#items \leq LIMIT$

INIT

$items = \langle\,\rangle$

Push

$\Delta(items)$

$item? : L$

$status! : \mathbb{Z}$

$\#items < LIMIT \Rightarrow status! >= 0 \land items' = \langle item? \rangle \frown items$

$\#items \geq LIMIT \Rightarrow status! < 0 \land items' = items$

$Pop \mathrel{\widehat=} [\,\Delta(items)\ item! : L \mid items = \langle item! \rangle \frown items'\,]$

RepStack[*X*] is then capable of being refined to utilise the definitions inherited by *RegularStack*[*X*]. As the features are identical, the refinement step is trivial:

RepStack[*X*]

$\lceil(Push, Pop, INIT)$

$RegularStack[X]$

The desired definition of *RepStack*[*X*] can be derived through refinement from the above definition, by strengthening the postcondition of the *Push* operation (by reducing non-determinism over the output variable *status*!).

RepStack[*X*]

$\lceil(Push, Pop, INIT)$

$RegularStack[X]$

Push

$\#items < LIMIT \Rightarrow status! = \#items'$

$\#items \geq LIMIT \Rightarrow status! = -LIMIT$

6 Completeness

In the previous sections we introduced three rules for structural refactoring of Object-Z specifications: Introduce Generic Parameter, Introduce Inheritance and Introduce Polymorphism. In [9], McComb also introduced

ANNEALING, which partitions a class state by introducing an instance to a fresh class definition. These four rules are minimal in the sense that they each operate upon one of the four orthogonal aspects of the object-oriented paradigm: no rule (or sequence of rules) can perform the function of any other rule.

We present an argument that the combination of these four rules with compositional class refinement [12] yields a complete method for design in Object-Z. That is, we demonstrate that it is possible to move from any structural design in Object-Z to any other design that represents a valid refinement of the original. However, we rely on some assumptions which we believe to be reasonable concerning the "system" class, i.e., the class describing the whole system (*System* in the example). First, we assume that the system class is not parameterised. In specifications where parameters exist over the system class, we expect these parameters to be instead expressed as given types in the specification. We also assume that the system class does not directly expose state variables via its visibility-list (although other classes may do this). This is because we cannot reason about the use of such variables, as the context of the system class is unknown, and this constrains possibilities for compositional class refinement. We expect accessor operations to be utilised in these circumstances.

Recall that each of the rules is an equivalence transformation that acts on the structure of a specification. We take advantage of this to demonstrate completeness by applying a commuting argument based on reduction. The reduction argument is as follows: if a sequence of rule applications exists that can reduce any arbitrary structure down to a canonical form, then it follows that the original structure can be *constructed* from that canonical form by the reverse application of the rules. By a commuting argument it follows that if a refinement ordering exists over the reduced form, then this ordering exists over arbitrary specifications, as all specifications are reducible to this form. Consequently, from any specification structure it is possible to construct another specification, with different structure, that is a valid refinement.

Figure 5 illustrates this schematically. A designer wishing to refactor a specification $(System, C_1, \ldots, C_n)$ to a specification $(System', D_1, \ldots, D_m)$ that is a refinement of or equivalent to $(System, C_1, \ldots, C_n)$ may do so by reducing $(System, C_1, \ldots, C_n)$ to two classes $(System, Delegate)$ (our reduced form); refining *System* and/or *Delegate* to derive $(System', Delegate')$; and then constructing $(System', D_1, \ldots, D_m)$ from $(System', Delegate')$.

We progress in five stages. The first stage applies an equivalence preserving refinement step to all classes that inherit features of other classes. This allows us

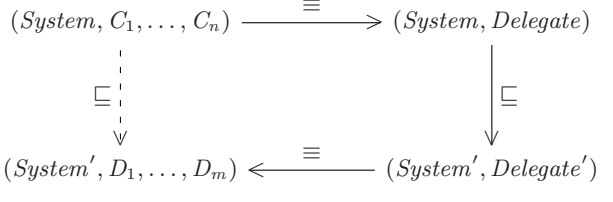


Fig. 5. Commuting argument for design reduction and construction with refinement

to establish the precondition for applying the INTRODUCE INHERITANCE rule in reverse. Through this process we effectively remove all inheritance relationships in the specification. The second stage introduces a *Delegate* class that is key to our reduced form. The third stage applies the INTRODUCE GENERIC PARAMETER rule in reverse. The precondition for applying this rule reversal is satisfied by forward applications of the INTRODUCE POLYMORPHISM rule. Through this process we show that all generic parameterisation can be removed from the specification. In the fourth stage, we apply the INTRODUCE POLYMORPHISM rule in reverse to collapse the entire specification (except for the *System* class) into the *Delegate* class. Again, we use equivalence preserving class refinements to satisfy the preconditions for applying this rule backwards. The last stage presents the argument that *System* and *Delegate* can be compositionally decoupled [12], and all possible specification refinements can be expressed as class refinements over these classes.

We refer to these stages respectively as: inheritance hierarchy flattening, interface isolation, parameterisation reduction, class paragraph definition reduction, and specification refinement. Below we expand on these stages to provide a high-level completeness argument; a formal proof can be found in [10].

1. Inheritance hierarchy flattening
   (a) Through equivalence preserving class refinements, references to features of superclasses can be removed.
   (b) Given subclasses do not reference features of their superclasses, the precondition for applying INTRODUCE INHERITANCE in reverse is satisfied for all inheritance relationships.
   (c) Repeated reverse application of INTRODUCE INHERITANCE removes all inheritance relationships.
2. Interface isolation
   (a) Since there are no inheritance relationships and the system class has no parameters, a fresh class definition $Delegate_1$ can be created via application of the ANNEALING rule on the system class [9]. ANNEALING partitions the state of a class into two parts: S and T, where T is the part that is moved into the newly created component class. In this case, the system class state is partitioned such that T represents the entirety of the state. That is, the entire state of the system class is moved to the new component class $Delegate_1$.
3. Parameterisation reduction (only necessary if classes with generic parameters exist)
   (a) For an arbitrary class definition that has generic parameters, all concrete types used to instantiate a parameter of that class can be enumerated.
   (b) For some class $C[P_1, \ldots, P_n]$ with a generic parameter P_i $(1 \leq i \leq n)$, an equivalence step can introduce a set of alias class names $AliasT_1[P_1, \ldots, P_n], \ldots, AliasT_m[P_1, \ldots, P_n]$ (axiomatically defined): one member for each different concrete type $T_1, \ldots, T_m$ used to instantiate P_i.
   (c) Since there are no inheritance relationships, the precondition for INTRODUCE POLYMORPHISM is satisfied for any class C and the alias classes $AliasT_1[P_1, \ldots, P_n], \ldots, AliasT_m[P_1, \ldots, P_n]$ from the previous step.

(d) Application of INTRODUCE POLYMORPHISM to each class C and its aliases establishes, in each case, the precondition for the reverse application of INTRODUCE GENERIC PARAMETER to each alias class for the chosen parameter P_i. An equivalence preserving data refinement to introduce a local definition of the form $L == X$ may be necessary.
(e) Reverse application of INTRODUCE GENERIC PARAMETER to all alias classes strictly reduces the number of generic parameters (by exactly 1).
(f) Repeating steps 3(a) through 3(e) removes all generic parameterisation for all classes, given that the system class is not parameterised.

4. Class paragraph definition reduction
   (a) Let $n \leftarrow 1$.
   (b) For any class definition C where C is not the system class or $Delegate_n$, both C and $Delegate_n$ can undergo an equivalence preserving class refinement such that each definition has identical features.
   (c) The precondition for the reverse application of INTRODUCE POLYMORPHISM is satisfied for a fresh class $Delegate_{n+1}$, which is defined to be the class union of C and $Delegate_n$.
   (d) Reverse application of INTRODUCE POLYMORPHISM to C and $Delegate_n$ yields a strict reduction in the number of classes.
   (e) Let $n \leftarrow n + 1$.
   (f) Repeating steps 4(b) through 4(e) eventually yields the reduced form, i.e., the only classes are the system class and $Delegate_n$.
5. Specification refinement
   (a) The $Delegate_n$ class can be compositionally decoupled from the system class, such that both may be refined, assuming the system class does not expose state variables in its interface [12]. Refinements to the system class and $Delegate_n$ constitute all possible specification refinements.

7 Conclusion

We have emphasised the completeness of the approach for manipulation of architectural *structure* (that is, high-level design), in an object-oriented sense. Unless a light-weight approach is employed, whereby individual class specifications — having undergone sufficient data refinement — are implemented (and tested) informally, there is also potential for future work in providing a rigorous method for implementing specifications of individual classes in an object-oriented programming language. As Object-Z is not wide-spectrum, this would require an extension to the language, or a mapping of our method to another object-oriented specification language that supports low-level programming constructs.

In ongoing work, Ruhroth et al. are investigating refactoring transformations of Object-Z specifications in the presence of CSP [13], particularly with respect to those refactoring transformations we refer to as non-structural. Other work by Estler et al. incorporates model-checking technologies for the verification of refactorings in Object-Z without CSP [4]. The automation of such verifications are important for the practicality and relevance of our approach to current software engineering practice, and can hopefully be extended to our structural rules.

Acknowledgements

We acknowledge the support of Australian Research Council (ARC) Discovery Grant DP0558408.

References

1. Java 2 Platform Standard Edition 5.0, `http://java.sun.com/j2se/1.5.0/guide/`
2. Borba, P., Sampaio, A., Cavalcanti, A., Cornelio, M.: Algebraic Reasoning for Object-Oriented Programming. Sci. Comput. Program. 52(1-3), 53–100 (2004)
3. Derrick, J., Boiten, E.: Refinement in Z and Object-Z: Foundations and Advanced Applications. FACIT Series. Springer, Heidelberg (2001)
4. Estler, H.-C., Ruhroth, T., Wehrheim, H.: Modelchecking correctness of refactorings – some experiments. ENTCS 187, 3–17 (2007)
5. Gheyi, R., Borba, P.: Refactoring Alloy specifications. ENTCS 95, 227–243 (2004)
6. Jackson, D.: Alloy: a lightweight object modelling notation. Software Engineering and Methodology 11(2), 256–290 (2002)
7. Lano, K.: Formal Object-oriented Development. Springer, Heidelberg (1995)
8. Lano, K., Goldsack, S.: Refinement of Distributed Object Systems. In: Najm, E., Stefani, J.-B. (eds.) Proc. of Workshop on Formal Methods for Open Object-based Distributed Systems, pp. 99–114. Chapman and Hall, Boca Raton (1996)
9. McComb, T.: Refactoring Object-Z Specifications. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 69–83. Springer, Heidelberg (2004)
10. McComb, T.: Formal Derivation of Object-Oriented Designs. PhD thesis, The University of Queensland (2007)
11. McComb, T., Smith, G.: Architectural Design in Object-Z. In: Strooper, P. (ed.) ASWEC 2004: Australian Software Engineering Conference, pp. 77–86. IEEE Computer Society Press, Los Alamitos (2004)
12. McComb, T., Smith, G.: Compositional class refinement in Object-Z. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 205–220. Springer, Heidelberg (2006)
13. Ruhroth, T., Wehrheim, H.: Refactoring object-oriented specifications with data and processes. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 236–251. Springer, Heidelberg (2007)
14. Smith, G.: The Object-Z Specification Language. Kluwer Academic Publishers, Dordrecht (2000)
15. Stroustrup, B.: The C^{++} Programming Language, 3rd edn. Addison-Wesley Longman Publishing Co., Inc, Boston (1997)
16. Woodcock, J.C.P., Davies, J.: Using Z: Specification, Proof and Refinement. Prentice Hall International Series in Computer Science (1996)

Formal Modeling of a Generic Middleware to Ensure Invariant Properties*

Xavier Renault[1], Jérôme Hugues[2], and Fabrice Kordon[1]

[1] Université Pierre & Marie Curie, Laboratoire d'Informatique de Paris 6/MoVe
4, place Jussieu, F-75252 Paris CEDEX 05, France
xavier.renault@lip6.fr, fabrice.kordon@lip6.fr
[2] GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France
jerome.hugues@enst.fr

Abstract. The complexity of middleware leads to complex Application Programming Interfaces (APIs) and semantics, supported by configurable components in the middleware. Those components are selected to provide the desired semantics. Yet, incorrect configuration can lead to faulty middleware executions, detected late in the development cycle.

We use formals methods to tackle this problem. They allow us to find appropriate composition of middleware components and the use of their APIs, and to detect valid or faulty sequences. To provide reusable results, we modeled a canonical middleware architecture using Z.

We propose a validation scenario to verify middleware's invariants. We define invariants to exhibit inconsistent usage of these APIs. The specification has been checked with the Z/EVES [13] theorem prover.

1 Introduction

Middleware is now a central piece of many applications. Therefore, expectations about middleware reliability increase. To meet this goal, their architecture is being revisited, cleaned to exhibit structuring patterns. For example, TAO [12] or Zen [10] take advantage of Design Patterns to provide a safer design. Based on this architecture, the middleware can be tuned to operate dedicated policies (tasking management, memory allocation, etc.).

Yet, these complex assemblies of design patterns has never been formally proved in an easy and efficient way. Thus, components interactions may lead to faulty configurations lately detected. A way to tackle this problem is to define and check invariants [5]. This is a way to express stability conditions on software components. For example, one can use OCL [9] to declare such invariants.

Should the middleware architecture be formally specified and its invariants formally expressed, one can define use case scenarios to verify the system. We aim to model middleware's components to ensure their composition. We specify invariants (*e.g.* array size of some internal structures, unicity of identifiers) for each component and check if they are verified for the selected assembly.

* This work is funded in part by the ANR/RNTL Flex-eWare project.

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 185–200, 2008.

Z [13] is an algebraic specification language, based on the mathematical typed set theory. It has a schema notation, which allow one to specify structures in the system. It relies on a decidable type system, allowing one to automatically perform well-formedness checking (e.g. interface matching, resource usage).

In this paper we use Z to specify middleware services. We compose them as a middleware developer or user would do. Then we formally prove that query identifiers are consistent.

Section 2 sketches the use of formal methods for middleware. Then Section 3 presents the canonical middleware we selected. Section 4 details its modeling with Z, and section 5 shows how we prove important properties on the system.

2 Applying Formal Methods to Middleware

Middleware provides a set of mechanisms to support distribution functions. Its typical architecture is made of components, each of which supports a particular step in the processing of interactions. Interactions are supported by the exchange of messages between a *client* and a *server*, representing the caller and the callee.

Middleware's architecture is a set of components supporting the different steps of this interaction. The use of components (and their variations, implementing different configurations or policies) allows developers to tune or to select configurations/policies to configure and deploy a middleware that meet application requirements. A *middleware configuration* is defined as a set of components implementation selected to fulfill requirements.

So far, middleware are described through their components and the service they implement. Components are often described with semi-formal notation such as UML. These notations allow to express invariants (e.g. OCL in UML).

However, very few middleware specification is based on formal methods. This is a problem because formal specification of components and their related properties is needed to formally ensure that invariants are still valid for a given configuration (*e.g.* the selected implementation of components respect the middleware invariants).

Related Works. There are two main approaches using formal methods in such context: dynamic and static.

Dynamic Verification deals with the system's behavior. It is the enumeration and analysis of all the states the system can reach during its execution. It is of interest to check if the system is deadlock or livelock free (model-checking). For example, in [11], LOTOS is used to build a formal framework to specify a middleware behavior based on its architecture [2]. Petri Nets [14] are also used to verify that PolyORB's core is both livelock and deadlock free [6].

Static Verification deals with structural aspects of a system and relies on predicate logic [5]. It is appropriate to analyse systems architectures, such as composition of interfaces from a typing theory point of view. It is also useful to check that invariants defined in the specification remains when components are composed. For example, in [1] the Z algebraic notation is used to verify the CORBA

[4] object model: this study exhibits inconsistencies in the OMG CORBA Security Services. Similar work has been achieved on the CORBA Naming Service [7] or on parts of the POSIX real-time standard [3]. They both characterize potential problems in the use of components (such as logical naming or messages typing). Z is also used in [8] to specify the architecture of a cognitive application for redesign.

However there is no approach that really deals with middleware architecture issues. They only describe high-level services or behavior.

Our Approach. We first analyse the architecture of a middleware to find relevant abstractions of the system. We also specify properties: 1) inside components, 2) at components interfaces, and 3) at the composition level.

The chosen architecture (presented in Section 3) is modular and versatile. It provides de facto a set of interesting abstractions for formal description. This work is complementary to the state of the art because we focus on the verification of properties for a given components assembly.

In an idealistic middleware development process, architecture verification appears after static verification (that deals with service specification) and before the dynamic verification (that ensures behavior of the system). It aims at building a middleware correct *by construction*. For example, we want to ensure that a configuration (i.e. a specific instantiation of selected components expressed by their formal specification) will not lead to a non-functionnal middleware (for instance one that cannot process requests).

3 Middleware Architectures

New architectures based on design patterns and genericity ease middleware adaptation by enhancing their configurability capabilities[12][10]. However their development process is not clearly specified and remains complex because it requires the implementation of most of the middleware functions.

We present in this section the characteristics of a specific middleware architecture we chose for our study.

The *Schizophrenic* architecture [15] is based on a Neutral Core Middleware (NCM), on which we can plug application-level and protocol-level personalities.

The Neutral Core Middleware (NCM) provides a set of canonical services on which we can build higher level services. The former are generic components for which a general implementation is provided. They are sufficient to describe various distribution models.

Application personalities are the adaptation layer between application components and middleware through a dedicated API or code generator. They register application components within the NCM; they interact with it to enable the exchange of requests between entities at the application-level.

Protocol personalities handle the mapping of personality-neutral requests (representing interactions between application entities) onto messages transmitted through a communication channel.

We chose the Schizophrenic architecture because the core of the middleware fits to formally specify services and invariants in the system: main services are grouped in this core, easing the formal analysis of the middleware.

PolyORB implements a schizophrenic architecture presented in [15]. It demonstrates the concept is sound: it implements multiple personalities like CORBA, GIOP, Distributed Systems Annex of Ada, a Message Oriented Middleware derived from Java's JMS and SOAP. From the NCM, we identify seven steps in the processing of a request, each of which is defined as a fundamental service. Their associated level is the middleware is depicted in Figure 1.

We present these fundamental services and their specific roles: first, the client looks up server's reference using the *addressing* service (a). Then, it uses the *binding* factory (b) to establish a connection with the server, using one communication channels (*e.g.* sockets, protocol stack). Request parameters are mapped onto a representation suitable for transmission over network, using the *representation* service (c) (e.g. CORBA CDR). A *protocol* (d) is implemented for transmissions between the client and the server nodes, through the *transport* (e) service; it establishes a communication channel between the two nodes. Then the request is sent through the network and unmarshalled by the server. Upon the reception of a request, the middleware instance ensures that a concrete entity is available to execute the request, using the *activation* service (f). Finally, the *execution* service (g) assigns execution resources to process the request.

Figure 1 also depicts the standard interaction between these fundamental services. It presents two hosts, each having the same underlying middleware: PolyORB. We present the main interactions between services from the export of a service from the server, to the sending of a request from the client. The object "Se" is called a *Servant* (from server side) or a Surrogate (object "Su"

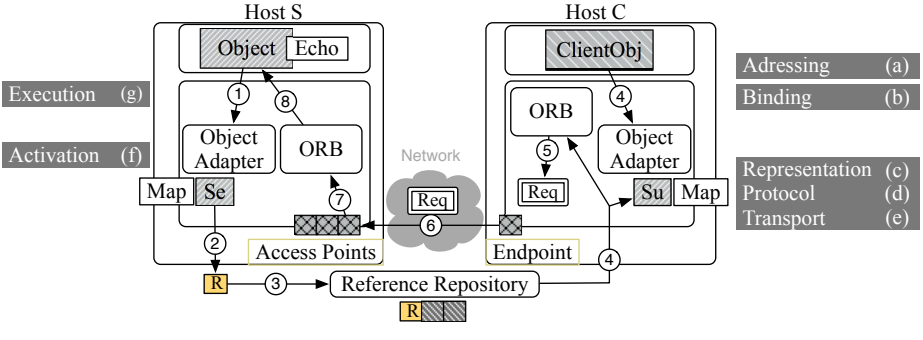


Fig. 1. Inside a schizophrenic middleware: PolyORB

from client side). It belongs to the Binding service and is, for the client, the local interface of the remote service.

This object is created (1) when an object *Object*, which belongs to an application personnality, wants to provide a service. It is registered in a map managed by an *Object Adapter* (OA), which is associated to an *Object Request Broker* (ORB). In order to share this service, a *Reference* R is built (2) and shared (3) to the system. A Reference contains information to locate and identify a service in the network. When a client object wants to invoke this service, the client ORB should get the specific reference (4) and extract from it a Surrogate, managed by the client OA. Then, a *Request* is built (5) from this Reference, and send through the local *Endpoint* managed by the client ORB. It uses protocol chosen from the Reference. This protocol belongs to a protocol personnality. The Request is received (6) through an *Access Point*, managed by the server ORB. The Request is then analysed (7): if it is valid, the Activation service is involved (8), using the OA, to select the local object which provides the service. Finally, the Execution service allocates needed resources.

4 Modeling Middleware with Z

We present the modeling in Z of the *Neutral Core Middleware* and related services, as presented in the previous section.[1]

The System. Several ORBs run into an heterogeneous system. At the specification level, the system is a collection of ORBs. Each ORB has to have unique name in this system. Furthermore, References on available services are shared through the system. The following Z schema specifies this system:

$$
\begin{array}{|l}
\hline
ORB\_System \\
orbs : \mathbb{F}\ ORB \\
ref\_repository : \mathbb{P}\ Reference\_Info \\
\hline
\#orbs \leq 1 \Rightarrow \\
\quad (\forall\, o1, o2 : ORB \\
\qquad \mid o1 \neq o2 \land \{o1, o2\} \subseteq orbs \\
\qquad \bullet\ o1.ORB\_TSAP \neq o2.ORB\_TSAP) \\
\hline
\end{array}
$$

A Z schema is divided into two parts (with an horizontal line as separator): the first one presents attributes names and type of a schema. The second presents invariants on these attributes. Schemas are a modular way to realize a specification, allowing one to split into small parts a system, and to compose them.

The invariant of the ORB_System schema states that if there is more than one ORB in the system, they must have different names (ORB_TSAP, see the definition of the ORB schema below).

[1] An extended version of this work is available at `http://pagesperso-systeme.lip6.fr/Xavier.Renault/pub/research/techreport/ZPolyORB_08.pdf`

The orbs set is theoretically infinite, but to specify the invariant related to names (which involves the cardinality), it is finite ($\mathbb{F}$ symbol). The ref_repository attribute is a infinite set of References. Each reference is unique.

Neutral Core Middleware Components. We present the main components of the NCM: the ORB and its associated Object Adapter.

The ORB manages several resources in the system: resources allocation, task scheduling, event priorities and request handlers. On one side, the ORB has to serve requests to applications entities, using an "Object Adapter". On the other side, it has to send and receive requests from other nodes, through Transport Access Points (TAP). Each TAP is bound to a specific protocol. Furthermore, the ORB is uniquely identified over the network. Hence the following Z schema:

ORB

$$
\begin{array}{l}
ORB\_TSAP : TSAPType \\
Request\_Queue : \mathrm{seq}\ Request \\
Transport\_Access\_Points : Transport\_Access\_Point \nrightarrow ProtocolStack \\
RootPOA : Object\_Adapter
\end{array}
$$

The ORB_TSAP attribute is the network identifier of each ORB.

Each ORB has a request queue, which is in our specification a sequence of Request. This allow us to keep the order of incoming requests, since it is an intrinsic properties of sequences in the Z notation.

The Transport_Access_Points attribute is the set of TAPs, each bounded to a specific ProtocolStack. It is a partial function of TAPs to ProtocolStack.

The Object Adapter (the RootPOA attribute) manages all objects which export services (named Servant), provided by the application. These servants have a unique identifier on their host. The Object_Adapter (OA) manages the mapping between these servants and their identifiers. It affects a unique identifier to each registered servant:

Object_Adapter

$$
\begin{array}{l}
Objects\_Map : ServantType \rightarrow Ident \\
\hline
\forall\, s : \mathrm{dom}\ Objects\_Map \\
\quad \bullet\ Objects\_Map(s) \neq NULLId
\end{array}
$$

In this Object_Adapter schema, Objects_Map models the binding between some servant to a unique identifiant. This is a total function between sets ServantType and Ident (symbolized by $\rightarrow$).

The invariant part of this schema states that it is not allowed to have entries in the map which have a NULLId (which means the Servant may not have been initialized for example).

A Reference is used to identify an application entity within the global system, as CORBA's IOR. It is a finite collection of Profiles. A profile carries the identifier of the target host in the system, and the identifier of the service within this host. It defines available protocols to contact the remote target, it is a subset of the protocols managed by both the client and the server.

$$\begin{array}{|l} \textit{Reference\_Info} \\ \hline \textit{Profiles} : \mathbb{F}(\textit{Profile}) \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{NULLRefSet} : \mathbb{P}\, \textit{Reference\_Info} \\ \hline \textit{NULLRefSet} = \\ \{r : \textit{Reference\_Info} \\ \quad \mid r.\textit{Profiles} = \emptyset\} \end{array}$$

$$\begin{array}{|l} \textit{Profile} \\ \hline \textit{TSAPName} : \textit{TSAPType} \\ \textit{ObjectId} : \textit{Ident} \\ \textit{Continuation} : \textit{ProtocolStack} \\ \hline \textit{ObjectId} \neq \textit{NULLId} \\ \hline \end{array}$$

$$\begin{array}{|l} \textit{NULLProfileSet} : \mathbb{P}\, \textit{Profile} \\ \hline \textit{NULLProfileSet} = \\ \{p : \textit{Profile} \mid \\ p.\textit{TSAPName} = \textit{NULLTSAP} \lor \\ p.\textit{Continuation} = \textit{NULLPStack}\} \end{array}$$

A Profile could not be bound to a servant whose identifiant is NULL. For specification purpose, we define the set of all null references (which have an empty set of Profiles) and the set of null profiles.

A Reference is built by a server host: a client has read-only access to this kind of resource: by construction, a Reference refers only to one service, and lists the different ways to reach it.

Neutral Core Middleware Services. To provide services as described in Section 3, different functions have to be defined in the middleware internals. We present them in the following order: first, from the server side, we define operations to share a service in the system; second, the functions used by a client to get information and send a request to the remote server; third, we present functions used by the server host to handle an incoming request.

From a server application, there are three mains steps to share and provide services in the network:

– The middleware exports the servant (Export procedure);
– The middleware produces a reference for this servant (Create_Reference);
– The middleware notifies other nodes about the availability of the service to the system, using a naming service for example.

Exporting a Service (Server Side). The Export procedure registers Servants in the Object Adapter (OA). This procedure should respect the fact that Servants are not managed twice in the OA map. Given a Servant, it creates a new entry in the map with an available identifier. This procedure fails if the Servant is already managed by the OA.

Export_OK

ΔORB_System
$O? : ORB$
$Obj? : ServantType$
$Oid! : Ident$

$O? \notin NULLORBSet$
$\wedge\ O? \in orbs$
$\wedge\ Obj? \neq NULLServant$
$\wedge\ Obj? \notin \text{dom}\ O?.RootPOA.Objects\_Map$
$Oid! \notin \text{ran}\ O?.RootPOA.Objects\_Map \wedge Oid! \neq NULLId$

$orbs' = (orbs \setminus \{O?\}) \cup \{\theta ORB[$
$\quad ORB\_TSAP := O?.ORB\_TSAP,$
$\quad RootPOA := \theta Object\_Adapter[$
$\quad\quad Objects\_Map := \{Obj? \mapsto Oid!\} \oplus O?.RootPOA.Objects\_Map],$
$\quad Transport\_Access\_Points := O?.Transport\_Access\_Points,$
$\quad Request\_Queue := O?.Request\_Queue]\}$

The Δ symbol indicates that Export_OK changes the state of ORB_System. Modified attributes of ORB_System are decorated with a single quote, as orbs'.

A Z schema can also be parametrized: input variables are decorated with a '?' symbol; output variables are decorated with the '!' symbol.

The θ symbol indicate an instantiation of a Z schema with specific values, affected by the ':=' symbol. Here, the Export_OK operator removes the input ORB variable from the system and inserts a new one with modified attributes. The intrinsic type of a total function $X \rightarrow Y$ is a set of pairs $\mathbb{P}(X \times Y)$. The $\oplus$ symbol adds a pair (x, y) into the set if there is no already existing pair as (x, z). In the later case, it overrides the old pair.

Notice the name of the presented operator ends with the "_OK" suffix. In order to avoid undefined predicates in the specification, each operator schema is divided into two schemas: one specifying preconditions to a successfull application of the operator (name ending with "_OK"), and the other specifying the dual preconditions of the first schema (name ending with "_ERR"). The final operator is build as a combination of these two schemas, and is then called *robust*. All operators in our specification are robust, but for sake of clarity we only present the "_OK" part of each of them.

Creating a Reference (Server Side). As previously defined, a Reference contains all information to contact and identify a service in the system. It carries information on the different protocols available to contact the remote node. These protocol are bound to a particular Transport Access Point. To create a reference, we need to build all information for each TAP. We need to set the local identifier of the service, and the global identifier of the host in the system. Figure 4 presents the related operator schema.

Since PSet!.Profiles is a set of profile, we can use the Z notation to build a *set comprehension*: the pattern is $\{x : T \mid C \bullet P\}$, where x is of type T. For each

Create_Reference_OK

$O? : ORB$
$Oid? : Ident$
$PSet! : Reference\_Info$

$O? \notin NULLORBSet$
$Oid? \neq NULLId$
$\wedge\ Oid? \in \text{ran}\ O?.RootPOA.Objects\_Map$
$PSet!.Profiles = \{$
$\quad TAP : \text{dom}\ O?.Transport\_Access\_Points$
$\quad\quad \bullet\ \theta Profile[$
$\quad\quad\quad TSAPName := O?.ORB\_TSAP,$
$\quad\quad\quad ObjectId := Oid?,$
$\quad\quad\quad Continuation := O?.Transport\_Access\_Points(TAP)]\}$

Fig. 2. The Create Reference operator

x under the condition C, then the predicate P holds. Here, a Profile set is built as follow: for each TAP in the input ORB's TAP, an instantiation of a Profile schema is made with specific value.

This operator fails if no Servant is bound to the input local identifier.

Sharing a Service (Server Side). Sharing a service is to notify the system that a new service is available.

Broadcast_Reference_Ok adds the input reference to the system's references repository. Each ORB in the system may then access to this repository to retrieve specific references.

Broadcast_Reference_OK

ΔORB_System
$PSet? : Reference\_Info$

$ref\_repository' = ref\_repository \cup \{PSet?\}$

Sending a Request (Client Side). We present operations that a client has to do in order to send a request to a remote server. The client should 1) get a Reference on the targeted service; 2) build the Request; 3) select the appropriate Profile and associated Protocol; 4) send its request to the remote host.

The Reference may be get using the reference repository, or with alternative mecanisms such as IORs.

The Request is the structure containing information to invoke the remote service. It is sent through the network, and carries a Reference on the targeted server object and the identifier for this server service.

It is a simplified view of a request, which may contain more parameters and additional QoS information.

Request

$Target : Reference\_Info$
$Operation : OperationIDType$

Create_Request_OK

$Target? : Reference\_Info$
$Operation? : OperationIDType$
$Req! : Request$

$Req! = \theta Request[$
$\quad Target := Target?,$
$\quad Operation := Operation?]$

The Create_Request_OK operator acts as a request factory: given a reference and a service identifiant, it builds the appropriate request.

Request Send operation first selects the appropriate profile (and the associated protocol) to contact the remote host; then it sends this request using the selected profile, adding the request to the targeted ORB queue. The send operator fails if the Request is malformed.

Send_Request_OK

ΔORB_System
$Req? : Request$
$P? : Profile$

$Req? \notin NULLReqSet$
$\exists var : ORB$
$\mid var \in orbs$
$\wedge\ var.ORB\_TSAP = P?.TSAPName$
$\bullet\ orbs' = (orbs \setminus \{var\}) \cup \{\theta ORB[$
$ORB\_TSAP := var.ORB\_TSAP,$
$RootPOA := var.RootPOA,$
$Transport\_Access\_Points :=$
$\quad var.Transport\_Access\_Points,$
$Request\_Queue :=$
$\quad var.Request\_Queue \frown \langle Req? \rangle]\}$

Select_Profile_OK

$Req? : Request$
$P! : Profile$

$Req? \notin NULLReqSet$
$Req?.Target.Profiles \neq \emptyset$
$\exists p : Profile$
$\quad \mid p \in Req?.Target.Profiles$
$\quad \bullet\ P! = p$

The Select_Profile_OK operator specifies a simple selection algorithm: it randomly picks one Profile among available ones.

In the Send_Request_OK operator, an incoming request is added to the targeted ORB queue (using sequence concatenation operator $\frown$).

Receive and Process Requests (Server Side). When receiving a request, the ORB checks, using the Activation service, that the request is valid and the targeted object is managed by its Object Adapter. Then, the Execution Service allocates resources for request execution.

Process_OK

ΔORB_System
$O? : ORB$
$FOid! : Ident$
$var : Request$

$O? \notin NULLORBSet$
$\land\ O? \in orbs$
$\land\ O?.Request\_Queue \neq \langle\rangle$
$var = head\ O?.Request\_Queue$
$\exists\, p : Profile$
$\quad \mid p \in var.Target.Profiles$
$\quad \land\ p.ObjectId \in \text{ran}\ O?.RootPOA.Objects\_Map$
$\quad \bullet\ FOid! = p.ObjectId$

$orbs' = (orbs \setminus \{O?\}) \cup \{\theta ORB[$
$\quad ORB\_TSAP := O?.ORB\_TSAP,$
$\quad RootPOA := O?.RootPOA,$
$\quad Transport\_Access\_Points := O?.Transport\_Access\_Points,$
$\quad Request\_Queue := tail\ O?.Request\_Queue]\}$

This operator dequeues an incoming request (using sequence operator like "head" and "tail", which respectively return the first element of a sequence and all the sequence but the first one).

Find_Servant_OK

$OA? : Object\_Adapter$
$id? : Ident$
$s! : ServantType$

$OA? \notin NULLOASet$
$\land\ id? \neq NULLId$
$\land\ id? \in \text{ran}\ OA?.Objects\_Map$
$\exists\, ss : ServantType$
$\quad \mid ss \in \text{dom}\ OA?.Objects\_Map$
$\quad \land\ OA?.Objects\_Map(ss) = id?$
$\quad \bullet\ s! = ss$

Find_Servant_ERR

$OA? : Object\_Adapter$
$id? : Ident$
$s! : ServantType$
$E! : Exception$

$OA? \in NULLOASet$
$\lor\ id? = NULLId$
$\lor\ id? \notin \text{ran}\ OA?.Objects\_Map$
$s! = NULLServant$
$E! = FAILURE$

The Find_Servant operator checks if the input identifier is related to a managed servant, and activates it. We have shown for the later operator the robust schema, including its "_OK" and "_ERR" parts. We present the robust Find_Servant operator (a combination of the two prevous ones):

$$Find\_Servant \mathrel{\widehat{=}} (Find\_Servant\_OK \land Success) \lor Find\_Servant\_ERR$$

5 Verifying Invariants in the Middleware Specification

We have defined the main components of a middleware. For the sake of place, this section presents only one system configuration, where only one client host and one server host are set. We present their associated initialization schema and then a scenario where the client host sends a oneway request to the server host. Such a request is sent with the "best-effort" semantic, for which the client host does not wait for any answer. In this scenario, we expose that some properties hold in the system, ensuring its consistency.

Our specification is checked using the Z/EVES theorem prover. Z/EVES is an interactive system for composing and analysing Z specifications. It helps the modeler to prove theorems on the specification, but for complex specification it only provides guidelines for proving: the modeler has to finish the proof and guide Z/EVES to get a "true" or "false" result.

System Instantiation. Both the client and the server hosts have an ORB. We present the initialization of one ORB, the server one. Since it is a transaction oriented architecture, server and client are identified only when one sends a request to the other. We introduce the global identifier of an ORB:

$$SHost : TSAPType$$

The definition of an Object Adapter sets initially its map to an empty set (no servant managed).

$$S\_OA : Object\_Adapter$$
$$\hline$$
$$S\_OA.Objects\_Map = \emptyset$$

We define, for this case study, only one Transport Access Point and its associated Protocol Stack for each ORB:

$$S\_TAP : Transport\_Access\_Point$$
$$S\_Protocol : ProtocolStack$$

We now instantiate an ORB, setting its name, Object Adapter, TAPs and Request_Queue:

$$S\_ORB : ORB$$
$$\hline$$
$$S\_ORB = \theta ORB[$$
$$\quad ORB\_TSAP := SHost,$$
$$\quad Request\_Queue := \langle\,\rangle,$$
$$\quad RootPOA := S\_OA,$$
$$\quad Transport\_Access\_Points := \{S\_TAP \mapsto S\_Protocol\}]$$

The previous initialization process holds for all hosts in the network. Once these hosts are set, we can initialize the whole environment: initializing the set of ORBs in the system and the references repository:

InitEnvironment

ORB_System'

$orbs' = \{S\_ORB\} \cup \{C\_ORB\}$
$ref\_repository' = \emptyset$

The system contains two running ORBs, with no request pending at the initialization time. Figure 5 introduces a Servant, named "Object", which will be managed by the server host for a transaction, and its associated service "Echo":

$Object : ServantType$ | $echo : OperationIDType$

Fig. 3. Z model of a Servant providing the Echo Service

Validation Scenario and Associated Proofs. We present a use-case scenario that corresponds to a typical activation of services. In this scenario, a server hosts an object which provides a service. The server application registers the object with its local middleware, creates a reference and shares it. A client host gets this reference, builds a request, and sends it to the remote entity. Finally, the server middleware handles the request.

As presented before, all operations that a server has to do in order to export a service are sequentialized as follow in a new schema Server_Op: it has to export the Servant to the Object Adapter, to create a reference on this Servant and to notify the system of the new service availability:

$$Server\_OP \mathrel{\widehat=} Export[O? := S\_ORB, Obj? := Object] \gg Create\_Reference[O? := S\_ORB] \gg Broadcast\_Reference\_OK$$

The $\gg$ symbol indicates operations are chained, where the output of one is the input of the other. These operations must be robust, to avoid undefined state.

In order to emit a request to the server host, a client should extract the remote object's reference, create a request targeting this object, adding request payload and select a profile to contact the remote host and finally send the request:

$$Client\_OP \mathrel{\widehat=} GetReference \gg Create\_Request[Target?, Operation? := echo] \gg Select\_Profile \gg Send\_Request$$

The GetReference schema is an operator defined in the scope of the case study. It is defined as:

GetReference

ΞORB_System

$Target! : Reference\_Info$

$Target! \in ref\_repository$

This schema is pretty simple since in our case study there is only one reference in the repository.

We restrict our case study to Oneway Request: for these later, the client does not wait for an answer. In this scenario, the server exports a service, the client sends a request, and the server checks this request before processing it.

$$ONEWAY \mathrel{\widehat=} InitEnvironment \fatsemi Server\_OP \fatsemi Client\_OP \fatsemi Process$$

The $\fatsemi$ symbol expresses a call sequence of operators.

In our scenario, we want to ensure that the request sent by the client contains the same id as the one exported by the server at the beginning.
To make this verification, we define the following test schema:

$$ONEWAYTest \mathrel{\widehat=} ONEWAY \gg Find\_Servant[OA? := S\_ORB.RootPOA, Foid?/id?]$$

The Process operator is invoked with the id sent with the request, and it returns the id of the activated object; we pipe it with the Find_Servant operator to check if it the same as Object . To verify this property, we express the theorem shown in Figure 4

theorem tOneWayReliable
$ONEWAYTest \Rightarrow s! = Object$

⟨... Z/EVES output ...⟩
Proving gives ...
true

Fig. 4. Theorem: a Oneway Request is reliable

Analysis For sake of clarity and readability, we do not present the whole interaction with Z/EVES. To achieve this proof, we needed to prove intermediates theorems, such as precondition reachability of each operators, schemas consistency and domain checking. We needed to set rules to help Z/EVES to finish the proof: typing related rules, transformation rules (predicate equivalence, etc.). Each rule has been proved in order to be used.

This global proof ensures that for this call sequence, invariants specified within each schema hold: names of ORBs are unique, no uninitialized objects are managed by Object_Adapters. Furthermore, preconditions for each operators are reachable and allow to produce valid postconditions as specified. These postconditions are checked and ensure that this combination of operators will lead to the seeked goal: identifiants consistency through a OneWay Request Process.

6 Conclusion and Future Work

In this paper, we presented the use of Z as formal notation to specify the architecture of a canonical middleware, based on a schizophrenic middleware architecture. This allows us to build abstraction of the middleware components, and to express properties and invariants upon each component of the system.

To elaborate the Z specification, we choose a well-structured architecture that relies on a canonical middleware core that concentrate all important services. Since these services are well-specified, it is possible to formally express them in Z. Moreover the execution path of these services is also well-identified by use-case scenarios that can serve as a basis for verification.

Once the canonical middleware core specified in Z, we have identified typical invariants for each components. These invariants are used to ensure that a given component configuration will not lead to inconsistencies in the middleware.

We experimented a well-identified use-case scenario on this architecture, and show its validity. Doing so with all use-case, we proved that our canonical architecture is consistent by construction.

One can enrich this specification, and add new contraints and invariants both deduced from a given implementation's caracteristics. Thus, our Z specification can serve as a framework to verify several variations based on our canonical middleware core.

The next step of our work is to express more invariants for each components, and to enrich the model with more details. We aim to analyse the impact of various QoS strategies on the middleware invariants. These QoS strategies will be expressed in Z to be bound to our current specification for analysis purpose. Categories of cases study will be defined and improved.

References

1. Basin, D., Rittinger, F., Viganò, L.: A Formal Analysis of the CORBA Security Service. In: Bert, D., P. Bowen, J., C. Henson, M., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 330–349. Springer, Heidelberg (2002)
2. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Comput. Netw. ISDN Syst. 14(1), 25–59 (1987)
3. Freitas, L.: Posix 1003.21 standard – real time distributed systems communication (in Z/Eves). Technical report, University of York (2006)
4. Object Management Group. Corba component model 4.0 specification. Specification Version 4.0, Object Management Group (April 2006)
5. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 26(1), 53–56 (1983)
6. Hugues, J., Thierry-Mieg, Y., Kordon, F., Pautet, L., Baarir, S., Vergnaud, T.: On the Formal Verification of Middleware Behavioral Properties. In: Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2004), Linz, Austria (September 2004)
7. Kreuz, D.: Formal specification of corba services using object-z. In: ICFEM 1998: Proceedings of the Second IEEE International Conference on Formal Engineering Methods, Washington, DC, USA, p. 180. IEEE Computer Society Press, Los Alamitos (1998)

8. Milnes, B., Pelton, G., Doorenbos, R., Laird, M., Rosenbloom, P., Newell, A.: A specification of the soar cognitive architecture in z. Technical report, Pittsburgh, PA, USA (1992)
9. OMG. OCL 2.0 Specification - Version 2.0 ptc/2005-06-06. OMG (June 2005)
10. Raman, K., Zhang, Y., Panahi, M., Colmenares, J., Klefstad, R., Harmon, T.: Rtzen: Highly predictable, real-time java middleware for distributed and embedded systems (2005)
11. Rosa, N., Cunha, P.: A formal framework for middleware behavioural specification. SIGSOFT Softw. Eng. Notes 32(2), 1–7 (2007)
12. Schmidt, D.C., Levine, D.L., Mungee, S.: The design of the TAO real-time object request broker. Computer Communications 21(4), 294–324 (1998)
13. Spivey, J.M.: The Z notation: a reference manual. Prentice-Hall, Inc., Upper Saddle River (1989)
14. Valk, R.: Basic definitions, ch. 4. In: Girault, C., Valk, R. (eds.) Petri nets and system engineering, 1st edn., pp. 41–51. Springer, Heidelberg (2003)
15. Vergnaud, T., Hugues, J., Pautet, L., Kordon, F.: PolyORB: a schizophrenic middleware to build versatile reliable distributed applications. In: Llamosí, A., Strohmeier, A. (eds.) Ada-Europe 2004. LNCS, vol. 3063, pp. 106–119. Springer, Heidelberg (2004)

CoBoxes: Unifying Active Objects and Structured Heaps

Jan Schäfer* and Arnd Poetzsch-Heffter

University of Kaiserslautern
{jschaefer,poetzsch}@informatik.uni-kl.de

Abstract. Concurrent programming in object-oriented languages is a notoriously difficult task. We propose coboxes – a novel language concept which combines and generalizes active objects and techniques for heap structuring. CoBoxes realize a data-centric approach that guarantees mutual-exclusion for groups of objects. This is important for more complex data structures with invariants ranging over several objects. CoBoxes support multiple entry objects, several cooperating tasks in a cobox, and nesting of coboxes for composition and internal parallelism. Communication between coboxes is handled by asynchronous method calls with futures, which is in particular suitable for distributed programming. In this paper, we explain how aspects of concurrent programming can be solved by coboxes. We develop a core language for coboxes and present a formal operational semantics for this language.

1 Introduction

Today's programming languages support concurrency mainly by multi-threading. Especially in object-oriented settings, multi-threaded programming is notoriously hard. The programmer has to control the shared state of threads which is difficult in object-oriented programming because state sharing is a dynamic property depending on the reference structure in the heap. Another problem for thread safety is the fact that programming invariants often depend on several objects. Thus, after violating an invariant by modifying some object X, a thread needs to work on other objects to reestablish the invariant before another thread can access X. As threads are based on preemptive scheduling and every thread may be suspended at any time leading to an arbitrary interleaving of threads, the programmer must explicitly prevent certain interleavings by using locking, a fairly primitive and error-prone programming construct.

Whereas preemptive scheduling causes problems within a single process, it is successful for processes on the operating system level. The reason is simple: processes do not share state and communicate via message passing. However, to substitute all method calls in OO programming by asynchronous message passing would throw out the baby with the bath water. The state space of a process would be restricted to a single object and the well-understood sequential

* Supported by the Deutsche Forschungsgemeinschaft (German Research Foundation).

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 201–219, 2008.

reasoning that can be used for thread-safe parts of programs with synchronous method calls could no longer be applied.

In this paper, we present a model that tries to combine the best of two worlds. It builds upon and integrates techniques for active objects [7] and for structuring the heap into groups of objects, so-called *boxes*, which have only been presented in a sequential setting [37] (*sequential boxes*). We call this symbiosis *concurrent boxes* or *coboxes*. Like sequential boxes, coboxes hierarchically partition the heap into runtime components consisting of multiple objects. To use coboxes, the programmer simply has to declare certain classes as cobox classes. Instantiating such a class creates a cobox together with its *main* object (also called *owner* object). A cobox may have multiple *tasks* which are scheduled cooperatively and where only one at a time may be active. Thus, within a cobox programs are executed sequentially with programmer-defined points of suspension and scheduling. CoBoxes run concurrently and communicate via message passing. To support composition of boxes and a restricted form of internal concurrency, coboxes can be nested, i.e. coboxes can contain other coboxes.

Contributions and Overview. This paper extends the sequential box concept that we developed in [37] to concurrency. The novel concurrency model generalizes active objects with asynchronous messages and futures to multiple object components. The resulting concurrent programming model enables the programmer to develop instantiable and composable components of scalable sizes. Although a cobox may expose several objects to the environment the model guarantees that tasks within a cobox are free of data-races and are executed atomically between suspension points.

After a discussion of related work, we explain and illustrate our programming model (Sect. 3). As central technical contribution, we present a formal small-step operational semantics of a core language for concurrent boxes and investigate some of its properties (Sect. 4). We conclude the paper after a discussion of the current limitations of our approach as well as possible solutions (Sect. 5).

2 Related Work and Motivation

Heap Structuring. A heap in OOLs is an unstructured graph of objects. The missing structure makes it difficult to reason about the behavior of object-oriented programs. Different approaches have been proposed to handle this problem. Ownership Types [11, 6, 34] and variants [2, 36, 38] statically structure the heap into ownership contexts. This can be used, for example, to ease program verification [13, 35] or to statically guarantee data-race and deadlock freedom in multi-threaded programs [5]. The concept of object ownership can also be encoded in a programming logic to achieve data-race [30] and deadlock [31] freedom by program verification. In a previous work, we used hierarchical structuring of the heap to modularly describe the behavior of sequential object-oriented runtime components [37].

Actors. The actor [1, 33] or active object model [7] treats every object as an actor. Actors run in parallel and communicate via asynchronous messages.

To allow result values for asynchronous method calls, futures [33] are used. In general, actors only have a single thread of control. This makes it difficult to implement multiple interleaved control flows within an actor, as this has to be explicitly implemented. This problem has been solved by Creol [32, 12], for example, which supports multiple control flows in objects. Another problem is the flat object model of actors. In general, actors cannot have a complex aggregate state. Some approaches allow actors to have local or passive objects [7, 8], which are deep copied between actors or cannot be referenced by other actors at all. To the best of our knowledge, no actor model has been proposed yet which allows multiple entry objects or treats hierarchical nesting.

Monitors. Monitors [20, 27] couple data with procedures, which are guaranteed to run in mutual exclusion. They support multiple control flows coordinated by condition variables. The original monitor concept has been designed without pointers and the aliasing problem [28] in mind. The realization of monitors in mainstream OO-languages typically has two shortcomings. It merely supports internal concurrency and only protects single objects.

Several Java modifications have been proposed to overcome these shortcomings. Guava [3] distinguishes between monitors and objects and allows monitors with complex aggregate state consisting of multiple objects, but without multiple entry objects for a single monitor. JAC [22] is an approach to specify concurrency mechanisms in Java in a declarative way, but only on the granularity of single objects. *Sequential Object Monitors* (SOM) [9] separate scheduling of requests from objects that handle requests. SOM does not allow multiple entry objects, nor multiple tasks in a SOM. *Parallel Object Monitors* (POM) [10] generalize SOM, by allowing multiple concurrently running threads in a single POM. In addition, multiple objects can be controlled by a single POM. In contrast to SOM, POM does not guarantee data-race freedom.

Transactions. When talking about (software) transactions one must distinguish the implementation side and the language side. There are many proposals how to implement transactions, either using ordinary locks [25], or implementing a transactional memory in software only [39], in hardware [23], or both [24]. A still open question is how transactional behavior could be integrated into object-oriented languages. Atomic blocks [21] are one idea, another are atomicity annotations [15]. The problem of these pure code-centric approaches is that still high-level inconsistencies can occur if invariants range over several objects [41]. Atomic sets [41] are one answer to overcome this problem. Communication of threads as well as other waiting conditions within transactions are addressed in [40]. Although transactions can be implemented distributively, transactional memory is not a concept which can easily be generalized to the distributed setting. Thus we see transactional memory as an orthogonal concept which may be integrated in our model to allow truly parallel tasks within a single cobox, for example. An interesting starting point would be the work of Smaragdakis et al. [40].

```
cobox class Consumer {
  Producer prod;
  Consumer(Producer p) { prod = p; }
  void consume() {
    Fut<Product> f = prod!produce();
    ... // do something else
    Product p = f.get; // wait for result
    ... // consume p
} }
```

```
interface Producer { Product produce(); }

cobox class SimpleProducer
  implements Producer {
  Product produce() {
    Product p = ... // produce p
    return p
  }
}
```

Fig. 1. Producer-Consumer example

3 Programming with Concurrent Boxes

The novel concept of our programming language are coboxes. Like objects, coboxes are runtime entities. CoBoxes are containers for objects. Every object exactly belongs to a single cobox for its entire lifetime. CoBoxes are also containers for cooperatively scheduled tasks, where at most one task is active and all other tasks are suspended. The important difference to other active object approaches is that activity is not bound to a single object, but to a set objects, namely the objects that belong to the same cobox. This also means that coboxes generalize active objects as active objects can be simulated by coboxes containing only single objects. All objects of a cobox can be referenced by other coboxes without any restriction. Moreover, coboxes can be nested, i.e. coboxes can be contained in other coboxes, which allows cobox-internal concurrency and reuse of cobox classes.

To explain our cobox concept we use a Java-like language called JCoBox. JCoBox is Java where the standard concurrency mechanisms are replaced by the following ones:

- Annotation of classes as cobox classes
- Asynchronous method invocation with futures as results
- Task coordination by await, get and yield expressions.

If none of these constructs is used, a JCoBox program behaves like a corresponding sequential Java program.

3.1 CoBoxes as Active Objects

If a cobox only consists of one object, it behaves like active objects in Creol [12]. We illustrate this with a simple producer-consumer example shown in Fig 1. Besides a Producer interface, the example contains a Consumer and SimpleProducer class, which both are declared as cobox classes. This means that objects of these classes always live in separate coboxes, as instantiating a cobox class creates a new cobox together with its *main* object. The Consumer object has a reference to a Producer object. To get a new Product, the consumer asynchronously invokes

```
cobox class PipelineProducer
  implements Producer {
  PreProducer pp = new PreProducer();
  FinalProducer fp = new FinalProducer();
  Product produce() {
    PreProduct p1 = pp!produce().await;
    Product p = fp!produce(p1).await;
    return p;
} }
```

```
cobox class PreProducer {
  PreProduct produce() { ... }
}

cobox class FinalProducer {
  Product produce(PreProduct p) {
    ...
} }
```

Fig. 2. Producer with a pipelined production

the produce method of the producer. That call immediately returns a future object of type Fut<Product>. While the producer is busy producing the product, the consumer can do something else in parallel. Eventually, the consumer uses get on the future object. This blocks the active task of the consumer until the future value is available, which is the case when the producer returns from the produce method.

Different coboxes can run concurrently. Thus, multiple consumers can concurrently access the same producer. The cobox semantics serializes such accesses. Each cobox manages an internal set of *tasks*. Whenever a method is invoked on an object of a cobox, a new task is created. At most one task is *active* in a cobox, all other tasks are *suspended*. An active task terminates if the execution of the invoked method terminates. It suspends itself by executing a yield statement or an await on a future. If no task is active, a scheduler selects one of the suspended tasks.

3.2 Nested CoBoxes and Internal Concurrency

To support concurrency within a cobox and to implement new cobox classes using existing ones, it is possible to create *nested* coboxes. A cobox is always nested in the cobox which created it. For example, the developer of the Producer class recognizes that the production process can be divided into two steps. The first step produces a PreProduct which can then be used in a second step to produce the final product. Thus, it has a production pipeline, which doubles the throughput of the producer. The pipelined producer implementation is shown in Fig. 2. The implementation of the produce method, asynchronously invokes the produce methods of the nested producers and uses await to wait for the result. Thus, the task that executes the produce method suspends itself after it has sent the first message and waits for the result. While the nested coboxes concurrently work on the production, the Producer can accept other produce messages and activate tasks, resulting in an interleaved execution of multiple tasks in the producer, as well as a parallel execution of the nested pre- and final producers. Notice that the use of nested coboxes is transparent to users of PipelinedProducer objects.

```
class LinkedList<V> { ... }

cobox class ProducerPool {
  LinkedList<Producer> prods;
  ProducerPool() {
    prods =
      new LinkedList<Producer>();
    // ... fill pool with producers
  }
  Producer getProducer() {
    return new ProducerProxy(this);
} }
```

```
class ProducerProxy implements Producer {
  ProducerPool pool;
  ProducerProxy(ProducerPool p) {
    pool = p;
  }
  Product produce() {
    while (pool.prods.isEmpty()) yield;
    Producer p = pool.prods.removeFirst();
    Product res = p!produce().await;
    pool.prods.append(p);
    return res;
} }
```

Fig. 3. Producer pool example

3.3 Multiple Entry Objects

The granularity of an object is often insufficient to model complex components. In general, a component has a complex internal state consisting of multiple objects as well as multiple objects which act as entry objects to the component, extending its external interface. For example, in the producer-consumer scenario, it might be useful to improve parallelism by introducing a pool of producers. The producer pool implementation is shown in Fig. 3. The ProducerPool cobox class has a field prods holding a list of currently free producers. A client can use the getProducer method to obtain a new Producer instance. The getProducer method returns an instance of a ProducerProxy. As the ProducerProxy class is not a cobox class, all its instances are contained in the ProducerPool cobox that created them. To better understand the cobox structure that appears at runtime, we show a runtime snapshot of the producer pool example in Fig. 4. It shows the nesting of the coboxes as well as which objects belong to which coboxes.

A ProducerPool is an example of a more realistic cobox. It consists of the main object of type ProducerPool, further ProducerProxy entry objects, the objects implementing the LinkedList, and the nested PipelinedProducer coboxes. The interesting part is the implementation of the produce method. First, it ensures that the list of free producers is not empty. This is done in a while loop, suspending the running task with a yield statement, until the list is not empty anymore. This illustrates a non-blocking wait. (So far, we did not include orthogonal concepts like guards or wait conditions to the language; cf. Sect. 5 for a discussion). As the active task always has exclusive access to all objects of its cobox until it explicitly suspends itself, the code in the produce method of class ProducerProxy can safely remove the first entry of the free producers list, as the list belongs to the same cobox as the ProducerProxy object. After removing the first element, the task asynchronously calls the produce method of the producer and suspends itself until the result is available by using the await operation. Finally, the used producer is appended to the list of free producers again.

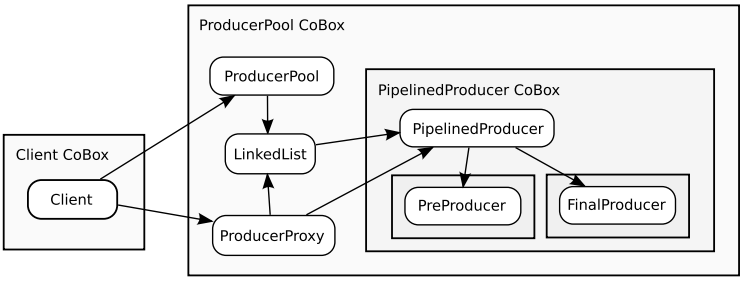


Fig. 4. A runtime snapshot of the producer pool example. Rectangles denote coboxes, rounded rectangles denote objects, edges with arrows denote references.

```
cobox class Subject {
  LinkedList<Observer> obs =
    new LinkedList<Observer>();
  State state = ... // initialize state
  State getState() { return state; }
  void updateState(State newState) {
    state = newState; notifyObservers();
  }
  void notifyObservers() {
    for (Observer o : obs) { o!changed(this); }
  }
  void register(Observer o) { ... }
}
```

```
cobox class Observer {
  void changed(Subject s) {
    // callback
    State state = s.getState();
    // ... do something
  }
}
```

Fig. 5. Subject-Observer example

3.4 Controlling Reentrant Calls

An important aspect of coboxes is that coboxes can control reentrant calls. Reentrant calls even happen in a purely sequential setting and complicate the reasoning about object-oriented programs. Figure 5 shows a typical subject-observer scenario. A Subject has an internal state and allows to register Observers which get notified about state changes. When notified, the Observer calls back to the Subject to get the new state. If this scenario would be implemented in sequential Java, the method call o.changed(this) would result in a reentrant call by the called observer. While the observers are notified about the state change, theoretically arbitrary calls to the Subject could happen. In the cobox implementation, the observers are notified by an asynchronous call. As the notifying task does not suspend itself, no other method can be activated in the cobox of the Subject until the task has finished. Thus, the s.getState() calls of the observers are delayed until all observers have been notified. This guarantees that the Subject cannot be changed while the observers are notified. Note that even though

we are in a concurrent setting, we get stronger guarantees than in the sequential Java setting, which eases behavioral reasoning.

The Java behavior can be simulated in JCoBox by calling await on the result of the o!changed(this) call, as in this case the current task suspends itself until the future value is available, i.e. until the called method has been finished. In between, other tasks can be activated, for example, tasks created by reentrant calls. Using a get instead of an await would result in a deadlock, because the active task waits for the result of the future without allowing other tasks to be activated – it *blocks* the cobox. Thus tasks created by reentrant calls cannot be activated. Note that this only applies if the future value is calculated by a task of a different cobox. If the future was the result of a self-call, that is, a call on an object of the same cobox, then such dependencies are resolved by activating the task responsible for calculating the future value.

4 Formal Semantics

In this section we present a formal dynamic semantics for a core language of coboxes, called JCoBox^C. JCoBox^C shares ideas from Featherweight Java [29], ClassicJava [16], Creol [12] and a previous formalization of a sequential language with boxes by the authors [37]. JCoBox^C only contains the core object-oriented language constructs for inheritance with method overriding and synchronous method calls as well as the new features for cobox classes, asynchronous method calls, futures, and task suspension. Other language features could be added as usual. Typing for JCoBox^C is similar to Java. The only necessary adaptation is the integration of futures which is straightforward.

4.1 Syntax

Figure 6 shows the abstract syntax of our language. Lower-case letters represent meta-variables, capital letters represent subsets of syntactic categories, and overbars indicate sequences of elements ($\bullet$ is the empty sequence, $\circ$ is concatenation of sequences). A program p is a set of class declarations D. To be executable a program must contain a cobox class Main with a main method. Such a program is then executed by creating an instance of Main and executing the main method. A class declaration d consists of a class name c, a super class name c', a list of field declarations $\overline{c\ f}$, and a set of method declarations H. We assume a predefined class Object with no fields, no methods and no super class. A class can be declared to be a cobox class by using the cobox keyword. Methods return the value of their body expression as result. New objects are created by the new c expression, all fields are initialized to null, constructors are not supported. let expressions introduce new local variables and can be used for sequential composition. An asynchronous method invocation, indicated by an exclamation mark, always results in a reference to a future. A future is an object of the special class Fut (details below). The value of a future can be obtained by either using get or await expressions with different blocking semantics. A yield suspends the currently active task allowing other tasks to proceed.

$$
\begin{aligned}
p \in \mathbf{Prog} &::= D \\
d \in D \subseteq \mathbf{ClassDecl} &::= [\mathsf{cobox}]\ \mathsf{class}\ c\ \mathsf{extends}\ c'\ \{\ \overline{c\ f};\ H\ \} \\
h \in H \subseteq \mathbf{MethDecl} &::= c\ n(\overline{c\ x})\{e\} \\
e \in E \subseteq \mathbf{Expr} &::= x \mid \mathsf{null} \mid \mathsf{new}\ c \mid e.f \mid e.f = e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \\
&\qquad e.n(\overline{e}) \mid e!n(\overline{e}) \mid e.\mathsf{get} \mid e.\mathsf{await} \mid \mathsf{yield} \mid e; e'
\end{aligned}
$$

$c \in$ class names, $n \in$ method names, $f \in$ field names, $x \in$ variable names

Fig. 6. Abstract syntax of JCoBox$^{\mathrm{C}}$

In the syntax above, we listed synchronous calls and sequential composition to stress that they are covered by the core language. However, in the following, we treat them as syntactic sugar to further compactify the semantics: $e.n(\overline{e}) \equiv e!n(\overline{e}).\mathsf{get}$ and $e; e' \equiv \mathsf{let}\ x = e\ \mathsf{in}\ e'$, where $x \notin FV(e')$.

4.2 Semantic Entities

A new aspect of the cobox semantics is that the heap is explicitly partitioned into the subheaps corresponding to coboxes. In particular, each cobox has its own objects. Compared to a formalization with a global heap and additional mappings capturing the box structure and the information to which box an object belongs, an explicit partitioning allows the creation of objects in a box-local way without knowing the global state. This simplifies modular treatment of box implementations (see [37]) and reflects distributed object-oriented programming.

The necessary semantic entities are shown in Figure 7. The state b of a cobox is represented by a tuple $\mathsf{B}\langle w, O, B, T, M, t_\epsilon, m_\epsilon\rangle$, consisting of a cobox reference w, a set of objects O, a set of nested coboxes B, a set of *suspended* tasks T, a set of incoming messages M, an optional *active* task t_ϵ and an optional current message m_ϵ. To simplify wording, we will not distinguish between a cobox and its state when it is clear from the context. A cobox reference is a sequence of cobox identifiers ι_b describing the path of the cobox in the nesting hierarchy of coboxes. This means that if a cobox has reference w then all its directly nested coboxes have references of the form $w.\iota_b$, whereas the *root* cobox has a single ι_b as reference.

Objects and their states are represented by triples $\mathsf{O}\langle \iota_o, c, \overline{v}\rangle$ with a box-unique object identifier ι_o, a class name c, and the list of field values $\overline{v}$. An object o always belongs to a certain cobox. If w is the reference of this cobox and ι_o the object identifier of o, the globally unique name of o is $w.\iota_o$.[1] The fields of an object can only be accessed by objects of the same cobox. Objects of other coboxes must use method calls.

Every cobox has a set of suspended tasks T and an optional active task t_ϵ. A task t corresponds to a single method activation and is represented by

[1] This is similar to the naming scheme in a tree-structured file system: coboxes correspond to directories, objects to files.

$$
\begin{array}{rcll}
b \in B \subseteq \mathbf{CoBox} & ::= & \mathsf{B}\langle w, O, B, T, M, t_\epsilon, m_\epsilon\rangle & \text{coboxes} \\
o \in O \subseteq \mathbf{Obj} & ::= & \mathsf{O}\langle \iota_o, c, \overline{v}\rangle & \text{objects} \\
 & | & \mathsf{O}\langle \iota_o, \mathsf{Fut}, v_\epsilon\rangle & \text{future objects} \\
t \in T \subseteq \mathbf{Task} & ::= & \mathsf{T}\langle r, e\rangle & \text{tasks} \\
m \in M \subseteq \mathbf{Msg} & ::= & \mathsf{M}\langle r, r', n(\overline{v})\rangle & \text{call messages} \\
 & | & \mathsf{M}\langle r, v\rangle & \text{return messages} \\
v \in V \subseteq \mathbf{Value} & ::= & r \mid \mathsf{null} & \text{values} \\
r \in R \subseteq \mathbf{Ref} & ::= & w.\iota_o & \text{object references} \\
w \in W \subseteq \mathbf{CoRef} & ::= & w.\iota_b \mid \iota_b & \text{cobox references} \\
e \in E \subseteq \mathbf{Expr} & ::= & \ldots \mid v & \text{extended expressions} \\
l \in \mathbf{Lab} & ::= & \uparrow m \mid \downarrow m \mid \tau & \text{labels} \\
\iota_o \in \mathbf{ObjId} & & & \text{object identifiers} \\
\iota_b \in \mathbf{BoxId} & & & \text{cobox identifiers}
\end{array}
$$

Fig. 7. Semantics entities of JCoBox$^{\mathrm{C}}$. Optional terms are indicated by an ϵ as index and may be ϵ. Small capital letters like B or K are used as "constructors" to distinguish the different semantic entities syntactically.

a pair $\mathsf{T}\langle r, e\rangle$ where r references the future that will receive the result of the call and where e is the expression to be reduced. CoBoxes communicate by exchanging messages. A message is either a call or return message. A call message $\mathsf{M}\langle r, r', n(\overline{v})\rangle$ consists of a future reference r, which waits for the result of the call, a receiver reference r' together with the called method name n, and argument values $\overline{v}$. A return message $\mathsf{M}\langle r, v\rangle$ consists of the target future reference r and the result value v.

Futures are modeled as objects of the predefined class Fut with the following implementation:

```
class Fut extends Object { Object await() { this.await } }
```

Instances of the Fut class cannot be created explicitly, but are implicitly created by an asynchronous method call. A future object has a single *optional* value v_ϵ, which is ϵ until the value of the future is available. This value cannot be set by field updates and may only be read by e.get or e.await expressions. Future references can be used like object references, in particular they can be passed to and used by other coboxes. Therefore, the Fut class contains an await() method that is not invoked explicitly, but used whenever a get or await operation is executed on a future object, not belonging to the current cobox. In that case, the await() method is asynchronously called on the future with an immediate get or await, respectively, resulting in a new future object, which acts as a proxy for the former future.

Evaluation Contexts. For a compact presentation we define some evaluation contexts [14]. An evaluation context is a term with a "hole" $\Box$ at a certain

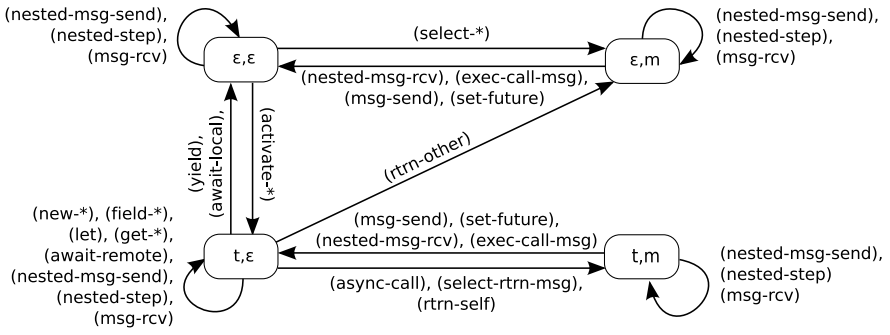


Fig. 8. Overview of the different reduction rules, showing the possible state transitions

position. By writing $e_\Box[e']$ that hole is replaced by term e'. In our syntax, holes can only appear at positions where expressions are expected.

$$\begin{aligned} e_\Box ::= {} & \Box \mid e_\Box.f \mid e_\Box.f = e \mid v.f = e_\Box \mid e_\Box.\mathsf{get} \mid e_\Box.\mathsf{await} \mid \\ & \mathsf{let}\ x = e_\Box\ \mathsf{in}\ e \mid e_\Box!m(\overline{e}) \mid v!m(\overline{v}, e_\Box, \overline{e}) \\ t_\Box ::= {} & \mathsf{T}\langle r, e_\Box \rangle \end{aligned}$$

4.3 Transition Rules

The dynamic semantics of JCoBox$^{\mathrm{C}}$ is defined in terms of a labeled transition system as a relation on coboxes and labels:

$$\longrightarrow \subseteq \mathbf{CoBox} \times \mathbf{Lab} \times \mathbf{CoBox}\,.$$

The notation $b \xrightarrow{l} b'$ means that cobox b can be reduced to b' with label l. We distinguish three kinds of transitions: internal steps ($l = \tau$), receiving a message ($l = \downarrow m$), and sending a message ($l = \uparrow m$). The transition rules are shown in Figures 10 and 12, which use auxiliary predicates and functions defined in Figures 9 and 11. To better understand in which order the different rules may be executed, we give an overview in terms of an automaton shown in Figure 8. The automaton states represent the currently active task and the current message of a cobox. The edges are labeled with the rules that can be applied. Objects, nested boxes, suspended tasks, and incoming messages are not shown in the automaton.

Expressions. We start by explaining how expressions are reduced, assuming that some task is active in the current cobox and that no current message exists (state t, ϵ). let expressions are handled by standard capture-avoiding substitution (LET). New objects of non-cobox classes are created in the current cobox with all values initialized to null (NEW-OBJ). If a class is annotated with cobox, a new nested cobox is created and the new object is created in that cobox instead of

$$\frac{\mathsf{cobox\ class}\ c\ \ldots}{cobox(c)} \qquad \frac{}{fields(\mathsf{Object}) = \bullet} \qquad \frac{\_\ \mathsf{class}\ c\ \mathsf{extends}\ c'\ \{\ \overline{c''\ f};\ H\ \}}{fields(c) = fields(c') \circ \overline{f}}$$

$$\frac{\_\ \mathsf{class}\ c\ \ldots\ \{\ \ldots\ \_\ n(\overline{\_\ x})\{e\}\ \ldots\ \}}{mbody(c,n) = (\overline{x})e} \qquad \frac{\_\ \mathsf{class}\ c\ \mathsf{extends}\ c'\ \{\ \ldots\ ; H\}\qquad n\ \text{not in}\ H}{mbody(c,n) = mbody(c',n)}$$

Fig. 9. Auxiliary predicates and functions extracting information from the (underlying) program code

the current cobox (NEW-COBOX). Field selects and field updates are only allowed on objects of the current cobox (FIELD-SELECT, FIELD-UPDATE). The rules select/update the ith value of the corresponding ith field and reduce to the old/new value. An asynchronous call is reduced to a reference of a new future object, which is added to the object set of the cobox. In addition, a call message is created and set as current message (ASYNC-CALL). A yield expression moves the active task into the set of suspended tasks, giving other tasks the possibility to be activated (YIELD).

Future Expressions. When a future reference r is used, two cases can be distinguished: *local futures* and *remote futures*. Local futures belong to the current cobox ($r = w.\iota_o$, where w is the reference of the current cobox), remote futures belong to a different cobox ($r \neq w.\iota_o$). A get expression on a local future is reduced to the value of the future if the value is available (GET-LOCAL-SUCCESS), otherwise the current task is blocked and remains active until the value becomes available. However, to prevent deadlocks, we have to treat synchronous self-calls as a special case. If a future is computed by a task of the same cobox ($t_\Box[w.\iota_o.\mathsf{get}] \in T$) then that task is activated, and the currently active task is suspended (GET-LOCAL-SELF) and later reactivated by (RTRN-SELF). Like a yield, an await on a local future suspends the currently active task (AWAIT-LOCAL). On a remote future reference r an await or get expression is transformed into an asynchronous method call of the await() method on that future with an immediate await or get, i.e. into a $r!await().\mathsf{await}$ or $r!await().\mathsf{get}$ expression, respectively (AWAIT-REMOTE, GET-REMOTE). These calls create corresponding local futures in the current cobox that are handled by the rules explained above.

Task Termination. A task terminates if its expression has been reduced to a single value. If the task was created due to a self-call and another task in the current cobox is blockingly waiting for the result ($t_\Box[r.\mathsf{get}] \in T$), that task is directly activated (RTRN-SELF). Together with rule (GET-LOCAL-SELF) this simulates a stack-like behavior for synchronous self-calls. If no task in the current cobox is blockingly waiting for the result, the active task is set to ϵ, i.e. it vanishes (RTRN-OTHER). In both cases a return message with the corresponding result value is set as current message. That message is treated in a next step by a message handling rule.

$$\frac{}{\mathsf{B}\langle w, O, B, T, M, t_\Box[\mathsf{let}\ x = v\ \mathsf{in}\ e], \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\Box[[v/x]e], \epsilon\rangle}\ (\textsc{let})$$

$$\frac{\neg \mathit{cobox}(c) \qquad \iota_o \text{ fresh in } O \qquad \overline{v} = \overline{\mathsf{null}} \qquad |\overline{v}| = |\mathit{fields}(c)| \qquad o = \mathsf{O}\langle \iota_o, c, \overline{v}\rangle}{\mathsf{B}\langle w, O, B, T, M, t_\Box[\mathsf{new}\ c], \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O \cup \{\, o \,\}, B, T, M, t_\Box[w.\iota_o], \epsilon\rangle}\ (\textsc{new-obj})$$

$$\frac{\overline{v} = \overline{\mathsf{null}} \qquad |\overline{v}| = |\mathit{fields}(c)| \qquad \mathit{cobox}(c) \qquad o = \mathsf{O}\langle \iota_o, c, \overline{v}\rangle \qquad \iota_b \text{ fresh in } B \qquad b = \mathsf{B}\langle w.\iota_b, \{\, o \,\}, \varnothing, \varnothing, \varnothing, \epsilon, \epsilon\rangle}{\mathsf{B}\langle w, O, B, T, M, t_\Box[\mathsf{new}\ c], \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B \cup \{\, b \,\}, T, M, t_\Box[w.\iota_b.\iota_o], \epsilon\rangle}\ (\textsc{new-cobox})$$

$$\frac{\mathsf{O}\langle \iota_o, c, \overline{v}\rangle \in O \qquad \mathit{fields}(c) = \overline{f}}{\mathsf{B}\langle w, O, B, T, M, t_\Box[w.\iota_o.f_i], \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\Box[v_i], \epsilon\rangle}\ (\textsc{field-select})$$

$$\frac{O = O' \uplus \{\, \mathsf{O}\langle \iota_o, c, \overline{v}\rangle \,\} \qquad \mathit{fields}(c) = \overline{f} \qquad O'' = O' \cup \{\, \mathsf{O}\langle \iota_o, c, [v/v_i]\overline{v}\rangle \,\}}{\mathsf{B}\langle w, O, B, T, M, t_\Box[w.\iota_o.f_i = v], \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O'', B, T, M, t_\Box[v], \epsilon\rangle}\ (\textsc{field-update})$$

$$\frac{\iota_o \text{ fresh in } O \qquad m = \mathsf{M}\langle w.\iota_o, r, n(\overline{v})\rangle \qquad o = \mathsf{O}\langle \iota_o, \mathsf{Fut}, \epsilon\rangle}{\mathsf{B}\langle w, O, B, T, M, \mathsf{T}\langle r', e_\Box[r!n(\overline{v})]\rangle, \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O \cup \{\, o \,\}, B, T, M, \mathsf{T}\langle r', e_\Box[w.\iota_o]\rangle, m\rangle}\ (\textsc{async-call})$$

$$\frac{t = t_\Box[\mathsf{yield}]}{\mathsf{B}\langle w, O, B, T, M, t, \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T \cup \{\, t \,\}, M, \epsilon, \epsilon\rangle}\ (\textsc{yield})$$

$$\frac{\mathsf{O}\langle \iota_o, \mathsf{Fut}, v\rangle \in O}{\mathsf{B}\langle w, O, B, T, M, t_\Box[w.\iota_o.\mathsf{get}], \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\Box[v], \epsilon\rangle}\ (\textsc{get-local-success})$$

$$\frac{t = t_\Box[w.\iota_o.\mathsf{get}] \qquad t' = \mathsf{T}\langle w.\iota_o, e\rangle}{\mathsf{B}\langle w, O, B, T \uplus \{\, t' \,\}, M, t, \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T \cup \{\, t \,\}, M, t', \epsilon\rangle}\ (\textsc{get-local-self})$$

$$\frac{t = t_\Box[w.\iota_o.\mathsf{await}]}{\mathsf{B}\langle w, O, B, T, M, t, \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T \cup \{\, t \,\}, M, \epsilon, \epsilon\rangle}\ (\textsc{await-local})$$

$$\frac{r \neq w.\iota_o}{\mathsf{B}\langle w, O, B, T, M, t_\Box[r.\mathsf{get}], \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\Box[r!\mathit{await}().\mathsf{get}], \epsilon\rangle}\ (\textsc{get-remote})$$

$$\frac{r \neq w.\iota_o}{\mathsf{B}\langle w, O, B, T, M, t_\Box[r.\mathsf{await}], \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\Box[r!\mathit{await}().\mathsf{await}], \epsilon\rangle}\ (\textsc{await-remote})$$

$$\frac{t_\Box[r.\mathsf{get}] \notin T}{\mathsf{B}\langle w, O, B, T, M, \mathsf{T}\langle r, v\rangle, \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, \epsilon, \mathsf{M}\langle r, v\rangle\rangle}\ (\textsc{rtrn-other})$$

$$\frac{t' = t_\Box[r.\mathsf{get}]}{\mathsf{B}\langle w, O, B, T \uplus \{\, t' \,\}, M, \mathsf{T}\langle r, v\rangle, \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t', \mathsf{M}\langle r, v\rangle\rangle}\ (\textsc{rtrn-self})$$

Fig. 10. Expression reduction rules

$$\frac{m = \mathsf{M}\langle r, v\rangle}{\mathit{isReturn}(m)} \qquad \frac{m = \mathsf{M}\langle r, r', n(\overline{v})\rangle}{\mathit{target}(m) = r'} \qquad \frac{m = \mathsf{M}\langle r, v\rangle}{\mathit{target}(m) = r}$$

Fig. 11. Auxiliary predicates and functions

(EXEC-CALL-MSG)
$$\frac{\mathsf{O}\langle \iota_o, c, \_\rangle \in O \qquad \mathit{mbody}(c, n) = (\overline{x})e \qquad t = \mathsf{T}\langle r, \mathsf{yield}; [w.\iota_o/\mathit{this}, \overline{v}/\overline{x}]e\rangle}{\mathsf{B}\langle w, O, B, T, M, t_\epsilon, \mathsf{M}\langle r, w.\iota_o, n(\overline{v})\rangle\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T \cup \{\, t \,\}, M, t_\epsilon, \epsilon\rangle}$$

(SET-FUTURE)
$$\frac{m = \mathsf{M}\langle w.\iota_o, v\rangle}{\mathsf{B}\langle w, O \uplus \{\, \mathsf{O}\langle \iota_o, \mathsf{Fut}, \epsilon\rangle \,\}, B, T, M, t_\epsilon, m\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O \cup \{\, \mathsf{O}\langle \iota_o, \mathsf{Fut}, v\rangle \,\}, B, T, M, t_\epsilon, \epsilon\rangle}$$

(ACTIVATE-WAITING-TASK)
$$\frac{\mathsf{O}\langle \iota_o, \mathsf{Fut}, v\rangle \in O}{\mathsf{B}\langle w, O, B, T \uplus \{\, t_\square[w.\iota_o.\mathsf{await}] \,\}, M, \epsilon, \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\square[v], \epsilon\rangle}$$

(ACTIVATE-YIELDED-TASK)
$$\frac{}{\mathsf{B}\langle w, O, B, T \uplus \{\, t_\square[\mathsf{yield}] \,\}, M, \epsilon, \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\square[\mathsf{null}], \epsilon\rangle}$$

(SELECT-CALL-MSG)
$$\frac{\neg \mathit{isReturn}(m)}{\mathsf{B}\langle w, O, B, T, M \uplus \{\, m \,\}, \epsilon, \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, \epsilon, m\rangle}$$

(SELECT-RTRN-MSG)
$$\frac{\mathit{isReturn}(m)}{\mathsf{B}\langle w, O, B, T, M \uplus \{\, m \,\}, t_\epsilon, \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\epsilon, m\rangle}$$

(MSG-SEND)
$$\frac{\mathit{target}(m) \neq w.r}{\mathsf{B}\langle w, O, B, T, M, t_\epsilon, m\rangle \xrightarrow{\uparrow m} \mathsf{B}\langle w, O, B, T, M, t_\epsilon, \epsilon\rangle}$$

(MSG-RCV)
$$\frac{\mathit{target}(m) = w.r}{\mathsf{B}\langle w, O, B, T, M, t_\epsilon, m_\epsilon\rangle \xrightarrow{\downarrow m} \mathsf{B}\langle w, O, B, T, M \cup \{\, m \,\}, t_\epsilon, m_\epsilon\rangle}$$

(NESTED-MSG-RCV)
$$\frac{b \xrightarrow{\downarrow m} b'}{\mathsf{B}\langle w, O, B \uplus \{\, b \,\}, T, M, t_\epsilon, m\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B \cup \{\, b' \,\}, T, M, t_\epsilon, \epsilon\rangle}$$

(NESTED-MSG-SEND)
$$\frac{b \xrightarrow{\uparrow m} b'}{\mathsf{B}\langle w, O, B \uplus \{\, b \,\}, T, M, t_\epsilon, m_\epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B \cup \{\, b' \,\}, T, M \cup \{\, m \,\}, t_\epsilon, m_\epsilon\rangle}$$

(NESTED-STEP)
$$\frac{b \xrightarrow{\tau} b'}{\mathsf{B}\langle w, O, B \uplus \{\, b \,\}, T, M, t_\epsilon, m_\epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B \cup \{\, b' \,\}, T, M, t_\epsilon, m_\epsilon\rangle}$$

Fig. 12. Message and task handling rules as well as nested box reduction rules

Tasks and Message Handling. Each task corresponds to a method activation. As method calls are initiated by call messages, a new task is created whenever the current message is a call message targeting an object of the current cobox (EXEC-CALL-MSG). The new task gets the reference of the future, which will hold

the return value and the body expression of the called method, where formal parameters are substituted by actual parameters. A yield statement is prepended, so that newly created tasks are treated like yielded tasks in the set of suspended tasks. If the current message is a return message targeting a future of the current cobox, the value of the future is set to the value of the return message (SET-FUTURE).

A cobox is *idle* if it has no active task and no current message (state ϵ, ϵ). In an idle cobox, a task can be activated if it either was suspended by a yield expression (ACTIVATE-YIELDED-TASK), or it was suspended by an await expression on a future whose value is available (ACTIVATE-WAITING-TASK). An idle cobox can as well select a message from the set of incoming messages and make it its new current message (SELECT-*). Whereas a call message can only be selected by an idle cobox, return messages can also be selected if a current task is active. This design decision reflects the goal to control all incoming calls, but allow return messages to reach their destination in nested coboxes without being blocked by active tasks in the surrounding box.

Messages are routed following the nesting structure. The parent coboxes are responsible for handling messages coming from and going to nested coboxes. This allows the parent cobox to control external communication of nested coboxes. If the current message is not targeting the current cobox, nor any nested cobox, it is *emitted* by rule (MSG-SEND). Messages can be received by a cobox if they target the cobox itself or any nested cobox (MSG-RCV). Messages can be received in any state and are added to the set of incoming messages. This reflects the asynchronous character of coboxes. If the current message targets a nested cobox, i.e. a nested cobox can receive that message, that message is send to the corresponding cobox by (NESTED-MSG-RCV). If a nested cobox emits a message that message is added to the set of incoming message of the parent cobox by (NESTED-MSG-SEND).

Internal steps of nested coboxes are propagated to its parent cobox and indirectly to the outermost cobox at the root of the box tree by (NESTED-STEP). Both rules are completely independent of the state of the current cobox and thus may be applied at any time, modeling concurrent behavior.

4.4 Program Execution

A program with a cobox class Main is executed by nesting a cobox instance b_{main} of Main in a special root or environment cobox b_{env} and setting the current message of b_{env} to a call message m_{main}, which invokes the main method of the main object ι'_o of b_{main}:

$$\begin{aligned} b_{env} &\equiv \mathsf{B}\langle \iota_b, \{\, \mathsf{O}\langle \iota_o, \mathsf{Fut}, \epsilon\rangle \,\}, \{\, b_{main} \,\}, \varnothing, \varnothing, \epsilon, m_{main}\rangle \\ b_{main} &\equiv \mathsf{B}\langle \iota_b.\iota'_b, \{\, \mathsf{O}\langle \iota'_o, \mathsf{Main}, \overline{\mathsf{null}}\rangle \,\}, \varnothing, \varnothing, \varnothing, \epsilon, \epsilon\rangle \\ m_{main} &\equiv \mathsf{M}\langle \iota_b.\iota_o, \iota_b.\iota'_b.\iota'_o, \mathsf{main}(\bullet)\rangle \end{aligned}$$

Program execution is performed by τ-transitions on the environment cobox:

$$b_{env} \xrightarrow{\tau} \cdots \xrightarrow{\tau} b'_{env}$$

When the main cobox finishes its execution of the main method, it will answer with a return message, whose result value is stored in the future object ι_o of the environment cobox.

4.5 Properties

JCoBox^C guarantees data-race freedom. As tasks can only access fields of objects of the same cobox and only one task in a cobox can be active, data-races can never occur. Furthermore, a task has atomic access to the objects of its cobox until it explicitly suspends itself.

Internal parallelism by nested coboxes is transparent to clients as all messages to nested coboxes are controlled by the parent cobox. This allows introducing parallelism within a cobox without changing its externally visible behavior.

A task can prevent reentrancy while waiting for a result value of a different cobox using get or synchronous method calls, as then the currently active task is not suspended. Consequently, no other task can be activated in the cobox. The only exception are tasks that have to be activated due to self-calls. Note again that reentrancy is not restricted to single objects, but can be handled for groups of objects.

A JCoBox^C program exactly behaves like a corresponding sequential Java program if no additional features of JCoBox^C are used and the only cobox is the initial Main cobox. This means that this cobox contains all objects that are created. As in sequential Java only synchronous method calls can be used, these calls are translated into asynchronous method calls with an immediate get. All these calls are self-calls, which means that tasks are created and executed in a stack-like way, simulating the call-stack of sequential Java.

5 Conclusions

We presented coboxes as a data-centric concurrency mechanism for object-oriented programs. CoBoxes hierarchically structure the object-heap into concurrently running multi-object components. CoBoxes may run completely parallel. Access to nested coboxes is controlled by the parent cobox. A cobox can have multiple *boundary* objects allowing the implementation of complex components with multiple entry objects. A cobox can have multiple tasks, which are scheduled cooperatively and only interleave at explicit release points. This guarantees that each task has exclusive access to the state of its cobox until it explicitly suspends itself. CoBoxes communicate via asynchronous method calls with futures, which allows modeling of synchronous method invocations as a special case. Our concurrency concept generalizes the active object model to hierarchically structured, parallel running groups of objects. It guarantees the absence of data-races and simplifies maintaining invariants ranging over groups of objects. We show how to use coboxes to write concurrent object-oriented programs and present a formal semantics for a Java-like core language with coboxes.

Current Limitations. The presented work is the first step towards generalizing the active object model to dynamically instantiable components supporting multiple internal and multiple boundary objects as well as component nesting. In the following we discuss current shortcomings and limitations.

In the given semantics, scheduling of incoming messages is nondeterministic. In practice, one could implement a default FIFO behavior, or allow the programmer to specify the scheduling algorithm [7, 9, 10]. Join patterns [17, 4, 18] could also be introduced as a scheduling and synchronization mechanism for incoming messages. Regarding coordination of tasks, in the presented model, tasks are activated nondeterministically when they are ready. A more fine-grained control is desirable, especially when waiting for conditions to become true. Concepts that can be used to achieve this are for example: guards [26], events [19], and condition variables [27].

A limitation of our current model is that objects and nested coboxes can only be created in the current cobox. We adopted this restriction to focus on the essential cobox concepts. More flexibility would be desirable, in particular the possibility to create objects or nested coboxes in other coboxes. One way to do this is to extend the `new` expression by an argument where the new object should be created.

Future Work. One of our future goals is to use the cobox model for distributed object-oriented programming such that each remote site corresponds to a cobox. In such scenarios, but also larger local systems, it is desirable to distinguish between value objects and objects passed by reference. Value objects are passed by creating a deep copy in the receiving cobox. This way the number of non-local messages can be drastically reduced. From a conceptual point of view, objects would be distinguished whether they represent data or not, like it is done in the Java RMI mechanism and other approaches [7, 3].

Acknowledgments. We thank Ina Schaefer and the anonymous reviewers for their helpful comments.

References

[1] Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge (1986)
[2] Aldrich, J., Chambers, C.: Ownership domains: Separating aliasing policy from mechanism. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 1–25. Springer, Heidelberg (2004)
[3] Bacon, D.F., Strom, R.E., Tarafdar, A.: Guava: a dialect of java without data races. In: OOPSLA 2000, pp. 382–400. ACM Press, New York (2000)
[4] Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C#. In: Magnusson, B. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 415–440. Springer, Heidelberg (2002)
[5] Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: OOPSLA 2002, November 2002, pp. 211–230. ACM Press, New York (2002)

[6] Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: POPL 2003, pp. 213–223. ACM Press, New York (2003)
[7] Caromel, D.: Towards a method of object-oriented concurrent programming. Comm. ACM 36(9), 90–102 (1993)
[8] Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. In: POPL 2004, pp. 123–134. ACM Press, New York (2004)
[9] Caromel, D., Mateu, L.: Éric Tanter: Sequential object monitors. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 316–340. Springer, Heidelberg (2004)
[10] Caromel, D., Mateu, L., Pothier, G.: Éric Tanter: Parallel object monitors. Concurrency and Computation: Practice and Experience (to appear, 2007)
[11] Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA 1998, pp. 48–64. ACM Press, New York (1998)
[12] de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
[13] Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. Journal of Object Technology 4(8), 5–32 (2005)
[14] Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theorectical Computer Science 103(2), 235–271 (1992)
[15] Flanagan, C., Freund, S.N., Lifshin, M.: Type inference for atomicity. In: TLDI 2005, pp. 47–58. ACM Press, New York (2005)
[16] Flatt, M., Krishnamurthi, S., Felleisen, M.: A programmer's reduction semantics for classes and mixins. Formal Syntax and Semantics of Java 1523, 241–269 (1999)
[17] Fournet, C., Gonthier, G.: The reflexive CHAM and the join-calculus. In: POPL 1996, pp. 372–385. ACM Press, New York (1996)
[18] Haller, P., Cutsem, T.V.: Implementing joins using extensible pattern matching. Technical Report LAMP-REPORT-2007-004, EPFL (August 2007)
[19] Hansen, P.B.: Structured multiprogramming. Comm. ACM 15(7), 574–578 (1972)
[20] Hansen, P.B.: 7.2 Class Concept. In: Operation System Principles, pp. 226–232. Prentice-Hall, Englewood Cliffs (1973)
[21] Harris, T., Fraser, K.: Language support for lightweight transactions. In: Crocker Jr., R., G.L.S. (eds.) OOPSLA 2003, pp. 388–402. ACM Press, New York (2003)
[22] Haustein, M., Löhr, K.P.: Jac: declarative Java concurrency. Concurrency and Computation: Practice and Experience 18(5), 519–546 (2006)
[23] Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proc. Int. Symp. on Computer Architecture (1993)
[24] Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. In: Tarr, P.L., Cook, W.R. (eds.) OOPSLA 2006, pp. 253–262. ACM Press, New York (2006)
[25] Hindman, B., Grossman, D.: Atomicity via source-to-source translation. In: MSPC 2006, pp. 82–91. ACM Press, New York (2006)
[26] Hoare, C.A.R.: Towards a theory of parallel programming. In: Operating System Techniques, pp. 61–71. Academic Press, London (1972)
[27] Hoare, C.A.R.: Monitors: An operating system structuring concept. Comm. ACM 17(10), 549–577 (1974)
[28] Hogg, J., Lea, D., Wills, A., de Champeaux, D., Holt, R.: The Geneva convention on the treatment of object aliasing. SIGPLAN OOPS Messenger 3(2), 11–16 (1992)
[29] Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. TOPLAS 23(3), 396–450 (2001)

[30] Jacobs, B., Piessens, F., Leino, K.R.M., Schulte, W.: Safe concurrency for aggregate objects with invariants. In: Aichernig, B.K., Beckert, B. (eds.) SEFM, pp. 137–147. IEEE Computer Society, Los Alamitos (2005)
[31] Jacobs, B., Smans, J., Piessens, F., Schulte, W.: A statically verifiable programming model for concurrent object-oriented programs. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 420–439. Springer, Heidelberg (2006)
[32] Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6(1), 35–58 (2007)
[33] Lieberman, H.: Concurrent object-oriented programming in Act 1. In: Yonezawa, A., Tokoro, M. (eds.) Object-Oriented Concurrent Programming, pp. 9–36. MIT Press, Cambridge (1987)
[34] Lu, Y., Potter, J.: On ownership and accessibility. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 99–123. Springer, Heidelberg (2006)
[35] Lu, Y., Potter, J., Xue, J.: Validity invariants and effects. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 202–226. Springer, Heidelberg (2007)
[36] Müller, P., Poetzsch-Heffter, A.: A type system for controlling representation exposure in Java. In: Drossopoulou, S., Eisenbach, S., Jacobs, B., Leavens, G.T., Müler, P., Poetzsch-Heffter, A. (eds.) Formal Techniques for Java Programs, Technical Report 269–5, Fernuniversität Hagen (2000)
[37] Poetzsch-Heffter, A., Schäfer, J.: A representation-independent behavioral semantics for object-oriented components. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 157–173. Springer, Heidelberg (2007)
[38] Schäfer, J., Poetzsch-Heffter, A.: A parameterized type system for simple loose ownership domains. Journal of Object Technology 5(6), 71–100 (2007)
[39] Shavit, N., Touitou, D.: Software transactional memory. In: PODC, pp. 204–213 (1995)
[40] Smaragdakis, Y., Kay, A., Behrends, R., Young, M.: Transactions with isolation and cooperation. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Steele Jr., G.L. (eds.) OOPSLA 2007, pp. 191–210. ACM Press, New York (2007)
[41] Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL 2006, pp. 334–345. ACM Press, New York (2006)

VeriCool: An Automatic Verifier for a Concurrent Object-Oriented Language

Jan Smans, Bart Jacobs, and Frank Piessens

Katholieke Universiteit Leuven, Belgium

Abstract. Reasoning about object-oriented programs is hard, due to aliasing, dynamic binding and the need for data abstraction and framing. Reasoning about *concurrent* object-oriented programs is even harder, since in general interference by other threads has to be taken into account at each program point.

In this paper, we propose an approach to the automatic verification of concurrent Java-like programs. The cornerstone of the approach is a programming model, a set of rules, which limits thread inference to synchronization points such that one can reason sequentially about most code. In particular, programs conforming to the programming model are guaranteed to be data race free. Compared to previous incarnations of the programming model, our approach is more flexible in describing the set of memory locations protected by an object's lock. In addition, we combine the model with an approach for data abstraction and framing based on dynamic frames. To the best of our knowledge, this is the first paper combining dynamic frames and concurrency.

We implemented the approach in a tool, called VeriCool, and used it to verify several small concurrent programs.

1 Introduction

In recent years, multi-processor and multi-core computers have become a commodity. To leverage the power provided by these multi-processor machines within a single application, developers must resort to multithreading. However, writing correct multithreaded programs is challenging. First of all, the non-determinism caused by thread scheduling makes finding errors through testing much less likely. Moreover, even when anomalies show up, they can be hard to reproduce. Secondly, reasoning about concurrent programs is hard, since in general interference by concurrently executing threads has to be taken into account at each program point. In particular, when threads concurrently access a shared data structure, special care has to be taken to avoid data races.

A data race occurs when two threads simultaneously access a shared memory location, and at least one of these accesses is a write access. Developers typically consider data races to be errors, since races can lead to hard-to-find bugs, and because they give rise to counter-intuitive, non-sequentially consistent executions under the Java memory model [1].

A simple strategy to prevent data races is to enclose each field access $o.f$ within a **synchronized**(o) block. Although this strategy is safe and rules out

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 220–239, 2008.

non-sequentially consistent executions, it is rarely used in practice, since it incurs a major performance penalty, is verbose, and only prevents low-level races. Instead, standard practice is to only lock objects that are effectively shared among multiple threads. However, it is difficult to determine which objects are meant to be shared and what locations are protected by an object's lock based solely on the program text. Therefore, it is hard for a compiler to determine whether the synchronization performed by the program suffices to rule out data races.

In this paper, we propose a programming model (a set of rules) such that programs that conform to the model contain no data races. In addition, we define a set of annotations to make the use of the programming model explicit. For example, a developer can annotate his code to make explicit whether an object is meant to be shared or not. Moreover, we explain how based on these annotations one can modularly and automatically verify whether a given program conforms to the model.

In summary, the contributions of this paper are as follows:

- We propose an approach to the automatic verification of concurrent Java-like programs. The cornerstone of the approach is a programming model for preventing data races. Compared to previous incarnations of the programming model [2,3,4], the approach is more flexible in describing the locations protected by an object's lock. The additional flexibility allows us to verify programs which cannot be verified in [2,3,4].
- To support data abstraction and framing, the approach relies on an existing solution based on dynamic frames [5,6]. A key insight of this paper is that dynamic frames can not only be used for abstract framing, but also to abstract over the locations protected by an object's lock. To the best of our knowledge, this is the first paper that combines concurrency and the dynamic frames approach.
- We implemented the approach in a tool, and used it to verify several concurrent programs. The verifier can be downloaded from the authors' homepage [7].

The remainder of this paper is structured as follows. In Section 2, we describe a programming model that ensures data race-freedom, and a way to verify whether a given Java program conforms to the model. In Section 3, we extend the approach of Section 2 with support for data abstraction. Finally, we compare with related work and conclude in Sections 4 and 5.

2 Preventing Data Races

In this section, we present the programming model that rules out data races (2.1), describe the annotations needed for modular verification (2.2), show how to statically verify whether an annotated program conforms to the model (2.3), and finally we explain why this verification approach works (2.4).

2.1 Programming Model

Our programming model prevents data races by ensuring that no two threads can ever access the same memory location concurrently. More specifically, we conceptually associate with each thread an access set, i.e. a set of memory locations that the thread can read and write, and the model guarantees that access sets of different threads are disjoint at all times. In our model, a location consists of an (object reference, field name) pair, and the location corresponding to $o.f$ is denoted $\&o.f$. The access set of a thread can grow and shrink over time. More specifically, four operations can affect a thread's access set:

- **Object creation.** When a thread t creates a new object o, all locations corresponding to fields of o are added to t's access set. For example, consider the constructor of the class *Counter* of Figure 1. At the beginning of this constructor, the field *count* of the newly created *Counter* object is made accessible to the current thread, and therefore it is safe to assign to the field within the body of the constructor.
- **Object sharing.** In addition to the accessibility of each location, the programming model also tracks each object's *sharedness*. That is, the model

```
final class Counter {
  int count;

  monitorfootprint
    { &count };
  monitorinvariant
    acc(count) ∧ 0 ≤ count;

  Counter()
    ensures acc(count);
    ensures ¬this.shared;
    ensures this.count = 0;
  {
    count = 0;
  }
}
```

```
class Session implements Runnable {
  shared Counter counter;

  Session(Counter c)
    requires c.shared;
    ensures acc(counter);
    ensures ¬this.shared;
  { counter = c; }

  void run()
    requires acc(this.∗);
    requires ∀{Object o • ¬o.locked};
  {
    synchronized(counter){
      counter.count + +;
    }
  }
}
```

```
Counter c = new Counter();
share c;
new Thread(new Session(c)).start();
new Thread(new Session(c)).start();
```

Fig. 1. A small concurrent program. The main program creates a new *Counter* object c, shares it, and then creates two threads. Each of these threads increments the counter within a **synchronized** block.

distinguishes *unshared* from *shared* objects. An unshared object is not meant to be locked, and a program violates the model if it attempts to do so. Conceptually, an unshared object has no corresponding lock. A shared object on the other hand can be locked by any thread.

The set of locations protected by a shared object's lock is called that object's *monitor footprint*. Our approach does not force the lock of an object o to protect all of o's fields, and therefore o's monitor footprint does not necessarily contain all fields of o. An object's *monitor invariant* is a predicate over its monitor footprint. Immediately after acquiring an object's lock, one may assume that the monitor invariant holds, and vice versa when releasing the lock one must establish the monitor invariant. For example, the monitor invariant of a *Counter* object o (Figure 1) states that the location corresponding to the field $o.count$ is accessible, and that the field holds a positive value.

Initially, new objects are unshared. When the execution of a thread t encounters a **share** o; statement[1], o transitions from the unshared to the shared state, provided o is not shared yet, o's monitor footprint is accessible to t, and o's monitor invariant holds. In addition, the sharing thread t loses access to all locations in o's monitor footprint. This is the only way an unshared object can become shared. After sharing, a thread must lock o to gain access to the locations in o's monitor footprint. Once an object is shared, it can never revert to the unshared state.

- **Acquiring and releasing locks.** After sharing an object o, threads can attempt to acquire o's lock. Whenever a thread t acquires this lock (e.g. in Java when t enters a **synchronized**(o) block), o's monitor footprint becomes accessible to t. Moreover, t may assume that o's monitor invariant holds over the locations in o's monitor footprint immediately after the acquisition of o's lock. When t decides to release o's lock, the thread again loses access to all locations in o's monitor footprint. Moreover, the programming model enforces that t can only release the lock when the monitor invariant holds.

  Note that an object's monitor footprint can grow and shrink over time. In particular, the monitor footprint at the time of locking does not have to equal the footprint at the time of release. This way accessibility of locations can be transferred from one thread to another.
- **Thread creation.** Starting a new thread transfers the accessibility of the locations corresponding to fields of the receiver object of the thread's main method (i.e. the *Runnable* object in Java or the *ThreadStart* delegate instance's target object in .NET) from the starting thread to the started thread. For example, starting a new thread with the *Session* object of Figure 1 transfers accessibility of the object's *counter* field from the starting thread to the started thread.

The programming model described above is similar to the techniques used in various extensions of separation logic [8,9,10,11]. We discuss them in Section 4.

[1] The **share** statement is a special annotation which can appear in the body of a program. It is discussed in more detail in Section 2.2.

2.2 Annotations

Executions of programs that conform to the programming model contain no data races. However, without annotations it is difficult to verify whether a program conforms to the model. For example, in general it is undecidable to determine based on the program text whether an object is always shared or unshared at a given program point. In addition, we want to perform *modular* verification. That is, the correctness of a method implementation must not depend on implementation details of other methods and modules. Therefore, we propose a set of annotations that make the use of the programming model explicit and enable modular verification. We explain each annotation by means of the example of Figure 1.

Method Contracts. Each method has a corresponding method contract, consisting of a precondition and a postcondition. Both the precondition and the postcondition are boolean side-effect free expressions. The former define valid method pre-states, while the latter define valid method post-states. An expression is side-effect free if it contains no object or array creations, simple or compound assignments, and contains no method invocations[2]. Only parameters and the variable **this** may occur free in preconditions. Postconditions may additionally mention the variable **result**, denoting the return value of the method. Furthermore, postconditions may contain old expressions **old**(e), denoting the value of the expression e in the method's pre-state.

Class Contracts. Each class has a corresponding class contract, consisting of a monitor footprint and a monitor invariant. The monitor footprint of a class C is a side-effect free expression of type **set** which defines the set of locations protected by objects with dynamic type C. An expression of type **set** represent a set of memory locations. Each memory location is a (object reference, field name) pair, and the location corresponding to $o.f$ is denoted $\&o.f$. For example, the lock of a *Counter* object c protects the singleton { $\&c.count$ }. The monitor invariant of a class C is a boolean, side-effect free expression, and the programming model ensures that it can be assumed to hold on entry of a **synchronized**(o) block (where o has dynamic type C), provided the invariant is established again at the end of each synchronized block. For example, the monitor invariant of *Counter* states that the *count* field is accessible and that it holds a positive value. Only the variable **this** may occur free in class contracts. An omitted monitor footprint defaults to the empty set, while an omitted monitor invariant defaults to *true*.

Ghost State. The programming model tracks for each object whether it is shared or unshared, locked or unlocked, and for each location which thread is allowed to access the location. To express these properties, we introduce three special expressions. $o.shared$ indicates whether o is shared. Similarly, $o.locked$ indicates whether o is locked by the current thread. Hence, the second precondition of the

[2] In Section 3, we relax the definition of side-effect free expression by allowing invocations of pure methods.

method *run*, which is the first method executed by a new thread, states that the thread holds no locks. Finally, the boolean expression $\mathbf{acc}(o.f)$ denotes whether the location $\&o.f$ is accessible to the current thread. All three expressions can only be used within monitor invariants, preconditions and postconditions. In particular, they cannot appear within method implementations. This restriction ensures that annotations are erasable, i.e. that they can be omitted without affecting the execution of the annotated program. The expression $\mathbf{acc}(o.*)$ is syntactic sugar for stating that all locations corresponding to fields of o are accessible.

Share Statement. A developer can indicate that an unshared object o should transition to the shared state via the **share** o; statement. For example, the code snippet at the bottom of Figure 1 creates a new *Counter* object c and shares it. **share** is a ghost statement which only affects ghost state. As such, it can be erased without affecting the original program.

Shared Modifier. A field can be annotated with a **shared** modifier, indicating that it can only hold *null* or a reference to a shared object. The field *counter* of the class *Session* is an example of such a field. In the next section, we store information about the sharedness of fields in invariants, and we no longer need this modifier.

The program of Figure 1 is correctly synchronized, and the annotations enable the static verifier to prove this. In the next subsection, we explain how verification works.

2.3 Verification

Our verifier takes an annotated program as input and generates, via a translation into an intermediate verification language called BoogiePL [12], a set of verification conditions. The verification conditions are first-order logical formulas whose validity implies the correctness of the program. The formulas are analyzed automatically by satisfiability-modulo-theory (SMT) solvers. Our approach is based on a general approach described in [13]. In this subsection, we focus on novel aspects of our approach: namely the modeling of accessibility, lockedness and sharedness, the tracking of monitor footprints and monitor invariants, and the translation of statements and expressions to BoogiePL in a way that guarantees compliance with the programming model.

Notation. Heaps are modeled in the verification logic as maps from object references and field names to values. For example, the expression $h[o, f]$ denotes the value of the field f of object o in heap h. The function *wf* returns whether a given heap is well-formed, i.e. whether the fields of allocated objects point to allocated objects. H denotes the current value of the global heap. Allocatedness of objects is tracked by means of a special boolean field named *alloc*. The function *typeof* returns the dynamic type of a given object reference.

$\mathsf{Tr}^{E}_{h_1,h_2}[\![e]\!]$ denotes the translation of the side-effect free expression e to an equivalent first-order, BoogiePL expression, in a context where h_1 denotes the

current value of the heap and h_2 denotes the value of the old heap. Similarly, $\mathsf{Df}^E_{h_1,h_2}[\![e]\!]$ denotes the definedness of the side-effect free expression e, in a context where h_1 denotes the current value of the heap and h_2 denotes the value of the old heap. For example, the definedness of the expression x/y is given by $y \neq 0$. We will omit the value of the old heap when translating single-state expressions. Finally, $\mathsf{Tr}^S[\![s]\!]$ denotes the translation of the statement s to a number of BoogiePL statements.

Ghost State. The programming model tracks for each location whether it is accessible or not. In the verification logic, the accessibility of the location $\&o.f$ in a heap h is denoted $h[o, \mathit{accessible}][f]$. That is, $h[o, \mathit{accessible}]$ is a map from field names to booleans where each entry indicates whether the corresponding location is accessible. In addition, the programming model divides the set of object references into shared and unshared objects, and it further subdivides the set of shared objects into locked and unlocked objects. In the verification logic, an object is shared if $h[o, \mathit{shared}]$ is true. Similarly, an object is locked by the current thread if $h[o, \mathit{locked}]$ is true.

Monitor Footprints and Invariants. The programming model associates with each object a monitor footprint and a monitor invariant. To model this association, the verification logic contains two function symbols: *monitorfootprint* and *monitorinvariant*. For instance, $\mathit{monitorfootprint}(o)$ returns the set of locations protected by o's lock. To connect these function symbols to the class contracts, we introduce an axiom per class. More specifically, for each class C with monitor footprint F and monitor invariant I, the verification logic contains the following axiom:

$$\textbf{axiom } (\forall h, \mathit{this} \bullet \mathit{wf}(h) \wedge h[\mathit{this}, \mathit{alloc}] \wedge \mathit{this} \neq \mathit{null} \wedge \mathit{typeof}(\mathit{this}) = C \Rightarrow \mathit{monitorfootprint}(h, \mathit{this}) = \mathsf{Tr}^e_h[\![F]\!] \wedge \mathit{monitorinvariant}(h, \mathit{this}) = \mathsf{Tr}^e_h[\![I]\!]);$$

The class contract of a class C with monitor footprint F and monitor invariant I is well-formed only if the following conditions hold: (1) I is well-defined (i.e. $\forall h, \mathit{this} \bullet \mathit{wf}(h) \Rightarrow \mathsf{Df}^E_h[\![I]\!]$), (2) F is well-defined provided I holds, (3) F only contains accessible locations assuming I holds, (4) I only requires accessibility of locations in F assuming that I holds, and (5) both I and F are framed by F provided I holds, that is any heap modification outside of F does not affect the values of I and F.

Translation of Statements and Expressions. The programming model consists of a set of rules that together guarantee the absence of data races. To verify whether a program complies with the model, we have to update the ghost state tracked by the model after each statement, and check that statements do not violate the model. Figures 3, 5 and 6 of Appendices A and C contain the translation to BoogiePL for key statements and expressions. In this section, we shortly highlight two important aspects of the translation.

A central rule in the programming model is that threads can only access locations in their corresponding access set. To enforce this restriction, each field access (both read and write) is preceded by a proof obligation requiring $\&o.f$ to be accessible the current thread.

In general, interference by concurrently executing threads has to be taken into account at each program point. However, our programming model enforces that threads can only read and write accessible locations. Moreover, the accessibility of a location with respect to a certain thread t can only be changed by t itself. Therefore, we only need to take into account the effect of other threads at synchronization points. More specifically, in the translation to BoogiePL, the effect of other threads is modeled by havocing (i.e. assigning non-deterministically chosen values to) locations that are added or removed from a thread's access set at share and lock statements.

Method Contract Validity. Programming errors can not only show up in method implementations, but also in method contracts. To ensure that contracts are meaningful, we check that they are well-defined. For a method m with precondition P, postcondition Q and parameters $x_1, \ldots x_n$ in a class C, the following proof obligation is generated:

$$\textbf{assert } (\forall h_{old}, h, \mathit{this}, x_1, \ldots x_n \bullet \mathit{wf}(h_{old}) \wedge \mathit{wf}(h) \wedge h[\mathit{this}, \mathit{alloc}] \wedge h_{old}[\mathit{this}, \mathit{alloc}] \wedge \mathit{this} \neq \mathit{null} \wedge \mathit{typeof}(\mathit{this}) <: C \Rightarrow \mathsf{Df}^E_{h_{old}}[\![P]\!] \wedge (\mathsf{Tr}^E_{h_{old}}[\![P]\!] \Rightarrow \mathsf{Df}^E_{h,h_{old}}[\![Q]\!]));$$

2.4 Soundness

In order to prevent data races, (1) threads should only be able to access locations in their access set, and (2) thread access sets and the monitor footprints of shared, non-locked objects must partition the set of allocated locations.

Since each field access is guarded by an accessibility check (see Figures 3 and 5), property (1) follows immediately. We outline a proof by induction on the length of the execution trace for property (2). In the initial state, the set of allocated locations is empty, the main thread's access set is empty, and no objects are shared yet. Therefore, property (2) holds in the initial state. Now assume that property (2) holds in a state σ. We have to demonstrate that each statement s leading from σ to a state σ' preserves (2). We consider two cases. The other cases are similar.

- Assume that s is a **share** o; statement. Successful verification ensures that o is non-null, unshared and that its monitor invariant holds in state σ. The well-formedness of the class contract implies that o's monitor footprint contains only locations that are accessible to the current thread. Immediately after the share statement, o is shared and the current thread can no longer access the locations in o monitor footprint. Property (2) is preserved, since the access sets of other threads are not affected, the footprints of shared, non-locked objects are not affected, and the old access set of the current thread is split in the new access set and the locations in o's monitor footprint.
- Assume that s is a lock acquisition (**synchronized**(o)). The program verified successfully, and hence we may assume that o was shared but not locked by the current thread before entering the synchronized block. Moreover, the Java semantics guarantee that upon entering the synchronized block no other thread was holding o's lock. As a consequence, we may assume that

o's monitor footprint is disjoint from any thread's access set in state σ. By adding the monitor footprint to the current thread's access set, and by making o locked, we preserve property (2) in σ'.

By induction, we may conclude that programs that verify only give rise to states where (2) holds.

3 Data Abstraction

Data abstraction is crucial in the construction of modular programs, since it ensures that internal changes in one module do not propagate to other modules. In object-oriented programs, classes typically enforce data abstraction by providing access to their internal state only through methods.

The class *Counter* in Figure 1 however was not written with data abstraction in mind, since it directly exposes its internal *count* field to client code. As a consequence, any change in *Counter*'s implementation forces us to rewrite or at least reconsider the correctness of client code. For example, renaming the *count* field breaks the implementation of *Session*'s *run* method.

Recently, we proposed an approach to the automatic verification of sequential, object-oriented programs that use method calls in specifications for data abstraction [5]. In this approach, methods used for this purpose have to be side-effect free, and are called pure methods. To solve the framing problem, that is to determine the effect of field assignments and executions of non-pure methods on the return values of pure methods, the approach relies on method footprint annotations, which specify an upper bound on the memory locations read (in case of pure methods) or written (in case of non-pure methods) by the corresponding method. More specifically, to prove that a state change (i.e. field update or non-pure method invocation) does not affect the return value of a pure method, one has to show that the footprint of the state change is disjoint from the pure method's footprint. Thanks to the use of dynamic frames [6], special pure methods that return a set of memory locations, method footprints can be specified without breaking data abstraction.

As an example, consider the class *Counter* of Figure 2. Contrary to the older version of the class of Figure 1, client code can only access the internal field *count* through the setter *increment* and the getter *getCount*. Furthermore, *Counter*'s method contracts do not mention the field *count*, and instead rely on public, pure methods to specify the behavior. In particular, both pure and non-pure methods define the locations they read or write in terms of the pure method *rep*, a dynamic frame. Since client code only depends on *Counter*'s public interface, changing the class's internal representation does not affect them. For example, renaming the field *count* would not endanger the correctness of the class *Session*.

In the approach described in the previous section, the monitor footprint of a class C specifies the set of locations protected by locks of objects with dynamic type C. For example, the class contract of *Counter* (Figure 1) specifies that the

lock of a *Counter* object o protects the singleton { $\&o.count$ }. However, this class contract exposes the internal field *count*. A key insight of this paper is that dynamic frames can not only be used to abstractly specify the locations read or written by a method, but also to abstract over the locations protected by an object's lock. Indeed, the dynamic frame *rep* can be used to specify *Counter*'s monitor footprint without revealing any internal fields, as shown in Figure 2. Similarly, the pure method *inv* can be used to abstractly specify the monitor invariant.

In summary, by combining the approach for verifying concurrent programs of the previous section with the solution for data abstraction and framing of [5], we can construct a verifier for concurrent programs that supports both data abstraction and framing. In the remainder of this section, we describe the extra annotations needed for dynamic frames, we highlight the most important changes in verification with respect to [5], and finally we sketch how our approach can be extended to deal with read-write locks.

3.1 Annotations

We extend the set of annotations with pure and predicate method modifiers and with method footprints.

Pure Methods and Predicates. A method can be annotated with a **pure** modifier, indicating that it can be used in specifications. The body of a pure method must consist of a single return statement returning a side-effect free expression. From now on, side-effect free expressions may contain method invocations but only to pure methods. Non-pure methods are called mutators. Pure methods with return type **set** are called dynamic frames. Purity is inherited by overriding methods.

Predicates are special pure methods marked with **predicate**. More specifically, a predicate is a boolean, pure method that can only be called from other predicates and within class and method contracts. Contrary to other method implementations, predicates are allowed to mention **acc**($o.f$), $o.shared$, and $o.locked$.

Method Footprints. In addition to pre and postconditions, each method contract contains a method footprint. A method footprint is a side-effect free expression of type **set**. The footprint of a pure method (**reads** annotation) specifies the locations that can potentially be be read by the method, while a mutator's footprint (**writes** annotation) specifies the locations that can be modified by the method. More specifically, a mutator can only modify $o.f$ if $\&o.f$ is in the method's footprint or if o was unallocated at the start of the method. Only parameters and the variable **this** may occur free in method footprints.

To prove that a state change (i.e. field update or non-pure method invocation) does not affect the return value of a pure method, one has to show that the footprint of the state change is disjoint from the pure method's footprint. To preserve disjointness of footprints, constructors and mutators should specify how

```
final class Counter {
  private int count;

  monitorfootprint rep();
  monitorinvariant inv();

  Counter()
    writes ∅;
    ensures inv();
    ensures getCount() = 0;
    ensures elemsFresh(rep());
    ensures ¬this.shared;
  { }

  void increment()
    requires inv();
    writes rep();
    ensures inv();
    ensures getCount() = old(getCount()) + 1;
    ensures newElemsFresh(rep());
  { count++; }

  pure int getCount()
    requires inv();
    reads rep();
  { return count; }

  predicate bool inv()
    reads inv()?rep() : universe;
  { return acc(count) ∧ 0 ≤ count; }

  pure set rep()
    requires inv();
    reads rep();
  { return { &count }; }
}
```

```
class Session implements Runnable {
  private Counter counter;

  Session(Counter c)
    requires c.shared;
    writes ∅;
    ensures inv();
    ensures elemsFresh(rep());
    ensures ¬this.shared;
  { counter = c; }

  void run()
  {
    synchronized(counter){
      counter.increment();
    }
  }

  predicate bool inv()
  { return acc(counter) ∧ counter ≠ null∧
      counter.shared; }

  pure set rep()
  { return { &counter }; }
}
```

```
Counter c = new Counter();
share c;
new Thread(new Session(c)).start();
new Thread(new Session(c)).start();
```

Fig. 2. A revision of the program of Figure 1. The classes *Counter* and *Session* now both hide their internal fields, and instead provide methods to query and update their state.

they affect footprints. To this end, we introduce two new boolean expressions: **elemsFresh**(e) and **newElemsFresh**(e). Both expressions can only be used in postconditions. **elemsFresh**(e) states that the locations in e are fresh with respect to the method pre-state, while **newElemsFresh**(e) states that each location in e is either an element of **old**(e) or is fresh with respect to the method pre-state.

The program of Figure 2 successfully verifies. The contracts for the classes *Runnable* and *Thread* which are needed for verification are shown in Figure 4 of appendix B. Note that the methods *run*, *inv* and *rep* in class *Session* override the corresponding methods in *Runnable*, and therefore they inherit the contracts of their overridden methods.

3.2 Verification

We shortly highlight the most important changes in the verification approach with respect to [5]. For details on the encoding of pure methods and method footprints, we refer the reader to [5].

In [5], a (postcondition) axiom is generated for each dynamic frame, stating that the set returned by the method only contains allocated locations. In this paper, we generate an additional axiom stating that dynamic frames only return sets containing accessible locations. This axiom is used to deduce that the monitor footprint of a newly acquired object does not overlap with any locations accessible to the thread before entering the synchronized block.

Furthermore, whenever a predicate method requires an object to be accessible, that is, whenever it contains a subexpression $\mathbf{acc}(o.f)$, we require $\&o.f$ to be an element of the method's reads clause. This allows predicates to be used in monitor invariants.

3.3 Read-Write Locks

A read-write lock is a variant of a traditional lock, which may be concurrently held by multiple reader threads, as long as there are no writers. For writer threads, the read-write lock is exclusive. For example, in the program of appendix D a shared arraylist object is protected by a read-write lock. Multiple threads can concurrently iterate over the elements in the list by acquiring the read lock ($\mathbf{synchronized}_R$). In previous incarnations of the programming model, it is impossible to verify this program.

Supporting read-write locks requires only minor extensions to the programming model and the annotations. More specifically, instead of associating an access set with each thread, we associate with each thread a read and a write set, the set of locations that can respectively be read or written by the corresponding thread. $\mathbf{acc}(o.f)$ now means that $\&o.f$ is in both sets, while $\mathbf{read}(o.f)$ signifies that $\&o.f$ is at least in the current thread's read set. Moreover, each class contract is extended with a read monitor footprint and a read monitor invariant. The regular monitor invariant must imply the read monitor invariant, and similarly the regular monitor footprint must be a superset of the read monitor footprint. When acquiring the read lock of an object o, the locations in o's read monitor footprint are added to the thread's read set, and one may assume o's read monitor invariant holds. And vice versa, when releasing a read lock, one must establish the read monitor invariant, and the thread loses read permission for the locations in the read monitor footprint.

4 Related Work

The Extended Static Checker for Java [14] (ESC/Java) is a compile-time program checker that attempts to find common errors, such as null dereferences and data races, in Java programs. Similarly to VeriCool, ESC/Java relies on verification condition generation and theorem proving. However, the tool trades soundness for ease of use. For example, it assumes that the value of a shared location stays unchanged if a method releases and then reacquires the lock that protects it, ignoring the possibility that some other thread might have acquired the lock and modified the location in the interim [15, page 91].

In his thesis, Kassios [6] describes a flexible approach to data abstraction and framing based on dynamic frames. More specifically, Kassios uses specification variables, similar to our pure methods, to achieve data abstraction. To solve the framing problem, he proposes using dynamic frames to abstractly specify the footprint of specification variables and the effect of mutator methods. Recently, we showed how Kassios' ideas can be incorporated in a program verifier for a Java-like language based on first-order logic [5]. However, both [6] and [5] consider only sequential object-oriented programs. In this paper, we extend the solution described in [5] in order to handle concurrent programs. To the best of our knowledge, this is the first paper combining dynamic frames and concurrency.

Various extensions of separation logic to concurrent programs have been proposed [9,10,16,8,11]. In particular, Gotsman *et al.* [10] propose a variant of concurrent separation logic that supports an unbounded number of dynamically allocated locks and threads, which is similar to our approach in many respects. For example, the boolean expression $\mathbf{acc}(o.f)$ can be considered to be the counterpart of the separation logic predicate $o.f \mapsto \_$ in the sense that they both represent a permission to access the location $\&o.f$. One difference between their approach and ours is that they allow lock finalization, while we never allow a shared object to become unshared. To achieve the same effect in our approach though, one could temporarily wrap the unshared object in a shared object. Another difference is that we make footprints explicit (typically via dynamic frames), while footprints are implicit in separation logic. Gotsman *et al.* developed a detailed formalization and soundness proof, but their approach has not been implemented in an automatic verifier.

Smallfoot [16,17] is an automatic verifier for concurrent separation logic geared toward verification of concurrent programs that manipulate recursive data structures. Smallfoot can verify many highly concurrent programs that our tool cannot. However, the tool relies on various built-in rules about list and trees. Our tool has no built-in rules to reason about particular data structures, but instead each class can define its own abstractions via pure methods.

Ábrahám-Mumm *et al.* [18] propose an assertional proof method for Java's reentrant monitors. Their approach supports class invariants, but these invariants can only mention fields of this. Our approach has no such restriction.

A number of type systems have been proposed to prevent data races in object-oriented programs. For example, Boyapati *et al.* [19] parametrize classes by the protection mechanism that will protect their objects against data races. The type system supports thread-local objects, objects protected by a lock, read-only object

and unique pointers. Our approach not only rules out data races, but also supports reasoning about richer properties such as object invariants.

The rules in our programming model that a lock's monitor invariant must hold whenever it is released and that it can be assumed to hold whenever it is acquired, are taken from Hoare's work on monitors [20].

This paper improves upon previous incarnations of the programming model [2,3,4]. First of all, the present approach is more flexible in specifying the locations protected by an object's lock. More specifically, [2,3,4] determine the contents of the monitor footprint by means of rep annotations on fields, i.e. all fields of all objects transitively reachable from an object o through rep fields are protected by o's lock. However, rep fields rule out sharing of memory locations (among different footprints). As a consequence, the concurrent iterator program of Figure 7 of appendix D, where different iterators share the locations in an array list's footprint, cannot be verified by [2,3,4]. Secondly, the present approach is more fine-grained in the sense that accessibility is tracked per location instead of per object. This implies that different fields of an object can be protected by different locks, which is not possible in previous versions. Finally, we demonstrate how to combine the programming model with an approach to data abstraction and framing based on dynamic frames. Data abstraction was not considered in [2,3,4].

5 Conclusion

We propose an approach to the automatic verification of concurrent Java-like programs. The cornerstone of the approach is a programming model for preventing both low-level and high-level races. Compared to previous incarnations of the model, we are more flexible in specifying the locations protected by an object's lock. In addition, we combine the model with an approach for data abstraction and framing based on dynamic frames [6,5]. To the best of our knowledge, this is the first paper that combines dynamic frames and concurrency. We implemented our approach in a tool, and used it to automatically verify several small concurrent programs.

In the future, we plan to apply our approach in a larger case study.

Acknowledgments

Jan Smans is a research assistant of the Fund for Scientific Research - Flanders (FWO). Bart Jacobs is a postdoctoral fellow of the Fund for Scientific Research - Flanders (FWO).

References

1. Gosling, J., Joy, B., Steele, G., Bracha, G.: The java language specification, 3rd edn. (2005)
2. Jacobs, B., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. In: SAVCBS (2004)

3. Jacobs, B., Leino, K.R.M., Piessens, F., Schulte, W.: Safe concurrency for aggregate objects with invariants. In: SEFM (2005)
4. Jacobs, B., Smans, J., Piessens, F., Schulte, W.: A statically verifiable programming model for concurrent object-oriented programs. In: ICFEM (2006)
5. Smans, J., Jacobs, B., Piessens, F., Schulte, W.: An automatic verifier for java-like programs based on dynamic frames (2008)
6. Kassios, Y.: A Theory of Object Oriented Refinement. PhD thesis, University of Toronto (2006)
7. `http://www.cs.kuleuven.be/~jans/vericool`
8. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: ESOP (2008)
9. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoretical Computer Science 375(1-3) (2007)
10. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local reasoning for storable locks and threads. In: APLAS (2007)
11. Haack, C., Hurlin, C.: Separation logic contracts for a java-like language with fork/join. Technical Report 6430, INRIA (2008)
12. DeLine, R., Leino, K.R.M.: Boogiepl: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-, -70 (2005)
13. Leino, K.R.M., Schulte, W.: A verifying compiler for a multi-threaded object-oriented language. In: Marktoberdorf Summer School Lecture Notes (2006)
14. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. In: PLDI (2002)
15. Leino, K.R.M., Nelson, G., Saxe, J.B.: Esc/java user's manual. Technical Report SRC-TN-2000-002, Compaq Research Center (2000)
16. Calcagno, C., Parkinson, M., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In: SAS (2007)
17. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: FMCO (2005)
18. Ábrahám Mumm, E., de Boer, F.S., de Roever, W.P., Steffen, M.: Verification for java's reentrant multithreading concept. In: Nielsen, M., Engberg, U. (eds.) ETAPS 2002 and FOSSACS 2002. LNCS, vol. 2303, Springer, Heidelberg (2002)
19. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: OOPSLA (2002)
20. Hoare, C.: Monitors: An operating system structuring concept. cacm 17(10) (1974)

Appendix

A Translation of Expressions

$$
\begin{array}{l}
\mathsf{Tr}_h^E[\![e.f]\!] \equiv h[\mathsf{Tr}_h^E[\![e]\!], f] \\
\mathsf{Tr}_h^E[\![\mathbf{acc}(e.f)]\!] \equiv h[\mathsf{Tr}_h^E[\![e]\!], \mathit{accessible}][f] \\
\mathsf{Tr}_h^E[\![\{\ \&e.f\ \}]\!] \equiv \{\ (\mathsf{Tr}_h^E[\![e]\!], f)\ \} \\
\mathsf{Tr}_h^E[\![e.\mathit{shared}]\!] \equiv h[\mathsf{Tr}_h^E[\![e]\!], \mathit{shared}] \\
\mathsf{Tr}_h^E[\![e.\mathit{locked}]\!] \equiv h[\mathsf{Tr}_h^E[\![e]\!], \mathit{locked}] \\
\mathsf{Tr}_h^E[\![e.m(e_1,\ldots,e_n)]\!] \equiv \#C.m(h, \mathsf{Tr}_h^E[\![e]\!], \mathsf{Tr}_h^E[\![e_1]\!], \ldots, \mathsf{Tr}_h^E[\![e_n]\!])
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Df}_h^E[\![e.f]\!] \equiv \mathsf{Df}_h^E[\![e]\!] \wedge \mathsf{Tr}_h^E[\![e]\!] \neq \mathit{null} \wedge h[\mathsf{Tr}_h^E[\![e]\!], \mathit{accessible}][f] \\
\mathsf{Df}_h^E[\![\mathbf{acc}(e.f)]\!] \equiv \mathsf{Df}_h^E[\![e]\!] \wedge \mathsf{Tr}_h^E[\![e]\!] \neq \mathit{null} \\
\mathsf{Df}_h^E[\![\{\ \&e.f\ \}]\!] \equiv \mathsf{Df}_h^E[\![e]\!] \wedge \mathsf{Tr}_h^E[\![e]\!] \neq \mathit{null} \wedge h[\mathsf{Tr}_h^E[\![e]\!], \mathit{accessible}][f] \\
\mathsf{Df}_h^E[\![e.\mathit{shared}]\!] \equiv \mathsf{Df}_h^E[\![e]\!] \wedge \mathsf{Tr}_h^E[\![e]\!] \neq \mathit{null} \\
\mathsf{Df}_h^E[\![e.\mathit{locked}]\!] \equiv \mathsf{Df}_h^E[\![e]\!] \wedge \mathsf{Tr}_h^E[\![e]\!] \neq \mathit{null} \\
\mathsf{Df}_h^E[\![e.m(e_1,\ldots,e_n)]\!] \equiv \mathsf{Df}_h^E[\![e]\!] \wedge \mathsf{Df}_h^E[\![e_1]\!] \wedge \ldots \mathsf{Df}_h^E[\![e_n]\!] \wedge \\
\qquad \mathsf{Tr}_h^E[\![e]\!] \neq \mathit{null} \wedge \mathsf{Tr}_h^E[\![\mathit{Pre}_m\,[e/\mathbf{this}, e_1/x_1, \ldots, e_n/x_n]]\!]
\end{array}
$$

Fig. 3. Translation and definedness of expressions

B Contracts for Runnable and Thread

```
class Thread {
  Thread(Runnable target)
    requires target ≠ null ∧ target.inv();
    writes target.rep();
    ensures inv();
    ensures elemsFresh(rep());
    ensures ¬this.shared;
  void start()
    requires inv();
    writes rep();
  predicate bool inv()
    reads inv()?rep() : universe;
  pure set rep()
    requires inv();
    reads rep();
}
```

```
interface Runnable {
  void run()
    requires inv();
    requires ∀{Object o • ¬o.locked};
  predicate bool inv()
    reads inv()?rep() : universe;
  pure set rep()
    requires inv();
    reads rep();
}
```

Fig. 4. The contracts for the interface *Runnable* and the class *Thread*

C Translation of Statements

We assume expressions nested within statements are side-effect free.

$\mathsf{Tr}^S[\![x = \textbf{new } C(e_1, \ldots, e_n);]\!] \equiv$
  assert $\mathsf{Df}_h^E[\![e_1]\!] \wedge \ldots \mathsf{Df}_h^E[\![e_n]\!]$;
  havoc *newObject*;
  assume $newObject \neq null \wedge \neg H[newObject, alloc] \wedge typeof(newObject) = C$;
  assume $(\forall q, f \bullet (q, f) \in locationsOf(newObject) \Rightarrow \neg H[q, accessible][f])$;
  $oldH := H$; **havoc** H; **assume** $successor(oldH, H)$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \vee (q, f) \in locationsOf(newObject) \Leftrightarrow$
    $H[q, accessible][f])$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \Rightarrow oldH[q, f] = H[q, f])$;
  assume $(\forall q \bullet H[q, locked] \Leftrightarrow oldH[q, locked])$;
  assume $H[newObject, alloc]$;
  call $C.ctr(newObject, \mathsf{Tr}_h^E[\![e_1]\!], \ldots, \mathsf{Tr}_h^E[\![e_n]\!])$;
  $x := newObject$;

$\mathsf{Tr}^S[\![e_1.f = e_2;]\!] \equiv$
  assert $\mathsf{Df}_H^E[\![e_1]\!] \wedge \mathsf{Df}_H^E[\![e_2]\!]$;
  assert $\mathsf{Tr}_H^E[\![e_1]\!] \neq null$;
  assert $H[\mathsf{Tr}_H^E[\![e_1]\!], accessible][f]$;
  $H[\mathsf{Tr}_H^E[\![e_1]\!], f] := \mathsf{Tr}_H^E[\![e_2]\!]$;

$\mathsf{Tr}^S[\![\textbf{share } e;]\!] \equiv$
  assert $\mathsf{Df}_H^E[\![e]\!]$;
  $sharedObject := \mathsf{Tr}_H^E[\![e]\!]$;
  assert $sharedObject \neq null \wedge \neg H[sharedObject, shared] \wedge monitorinvariant(H, sharedObject)$;
  $oldH := H$; **havoc** H; **assume** $successor(oldH, H)$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \wedge \neg(q, f) \in monitorfootprint(oldH, sharedObject) \Leftrightarrow$
    $H[q, accessible][f])$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \wedge \neg(q, f) \in monitorfootprint(oldH, sharedObject) \Rightarrow$
    $oldH[q, f] = H[q, f])$;
  assume $(\forall q \bullet H[q, locked] \Leftrightarrow oldH[q, locked])$;
  assume $H[sharedObject, shared]$;

Fig. 5. Translation of statements

$\mathsf{Tr}[\![\textbf{synchronized}(e)\{\ s\ \}]\!] \equiv$
  assert $\mathsf{Df}_H^E[\![e]\!]$;
  $lockedObject := \mathsf{Tr}_H^E[\![e]\!]$;
  assert $H[lockedObject, shared] \wedge \neg H[lockedObject, locked]$;
  $oldH := H$; **havoc** H; **assume** $successor(oldH, H)$;
  assume $monitorinvariant(H, lockedObject)$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \vee (q, f) \in monitorfootprint(H, lockedObject) \Leftrightarrow H[q, accessible][f])$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \Rightarrow oldH[q, f] = H[q, f])$;
  assume $(\forall q, f \bullet (q, f) \in monitorfootprint(H, lockedObject) \Rightarrow \neg oldH[q, accessible][f])$;
  assume $(\forall q \bullet q \neq lockedObject \Rightarrow H[q, locked] \Leftrightarrow oldH[q, locked])$;
  assume $H[lockedObject, locked]$;
  $\mathsf{Tr}^s[\![s]\!]$
  assert $monitorinvariant(H, lockedObject)$;
  $oldH := H$;
  havoc H;
  assume $successor(oldH, H)$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \wedge \neg(q, f) \in monitorfootprint(oldH, lockedObject) \Leftrightarrow H[q, accessible][f])$;
  assume $(\forall q, f \bullet oldH[q, accessible][f] \wedge \neg(q, f) \in monitorfootprint(oldH, lockedObject) \Rightarrow oldH[q, f] = H[q, f])$;
  assume $(\forall q \bullet q \neq lockedObject \Rightarrow H[q, locked] \Leftrightarrow oldH[q, locked])$;
  assume $\neg H[lockedObject, locked]$;

Fig. 6. Translation of statements (cont.)

D Concurrent Iterator

```
final class ArrayList {
  readmonitorfootprint rep_R();
  readmonitorinvariant inv_R();

  int count; Object[] items;
  ...
  pure Object get(int i);
    requires inv_R() ∧ 0 ≤ i < size();
    reads rep_R();
  { return items[i]; }

  pure int size();
    requires inv_R();
    reads rep_R();
    ensures 0 ≤ result;
  { return count; }

  predicate bool inv_R()
    reads inv_R()?rep_R() : universe;
  { return read(count) ∧ read(items)∧
    items ≠ null ∧ read(items.elems)∧
    0 ≤ count ≤ items.length; }

  pure set rep_R()
    requires inv_R();
    reads rep_R();
  { return {&count, &items} ∪ elems(items); }
}
```

```
class Iterator {
  ArrayList list; int index;
  ...
  Object next()
    requires inv() ∧ hasNext();
    writes rep();
    ensures inv() ∧ list() = old(list());
    ensures newElemsFresh(rep());
  { return list.items[index++]; }

  pure bool hasNext()
    requires inv();
    reads rep() ∪ list().rep_R();
  { return index < list.count; }

  pure ArrayList list()
    requires inv();
    reads rep();
  { return list; }

  predicate bool inv()
    reads inv()?
      (rep() ∪ list().rep_R()) : universe;
  { return list ≠ null ∧ list.inv_R()∧
    0 ≤ index ≤ list.count∧
    &list ∉ list.rep_R()∧
    &index ∉ list.rep_R(); }

  pure set rep()
    requires inv();
    reads rep();
  { return {&list, &index}; } }
```

Fig. 7. The iterator pattern. By using read locks, multiple threads can simultaneously iterate over a shared array list.

```
class Session implements Runnable {
  private ArrayList list;

  Session(ArrayList l)
    requires l.shared;
    writes ∅;
    ensures inv();
    ensures elemsFresh(rep());
    ensures ¬this.shared;
  { list = l; }

  void run()
  {
    synchronized_R(list){
      Iterator iter = new Iterator(list);
      while(iter.hasNext())
        loopinvariant iter.inv() ∧ newElemsFresh(iter.rep())∧
          iter.list() = old(iter.list());
        writes iter.rep();
      { iter.next(); }
    }
  }

  predicate bool inv()
  { return acc(counter) ∧ counter ≠ null ∧ counter.shared; }

  pure set rep()
  { return { &counter }; }
}
```

Fig. 8. The iterator pattern (cont)

A Caller-Side Inline Reference Monitor for an Object-Oriented Intermediate Language

Dries Vanoverberghe* and Frank Piessens

Dries.Vanoverberghe, Frank.Piessens@cs.kuleuven.be

Abstract. Runtime security policy enforcement systems are crucial to limit the risks associated with running untrustworthy (malicious or buggy) code. The inlined reference monitor approach to policy enforcement, pioneered by Erlingsson and Schneider, implements runtime enforcement through program rewriting: security checks are inserted inside untrusted programs.

Ensuring complete mediation – the guarantee that every security-relevant event is actually intercepted by the monitor – is non-trivial when the program rewriter operates on an object-oriented intermediate language with state-of-the-art features such as virtual methods and delegates.

This paper proposes a caller-side rewriting algorithm for MSIL – the bytecode of the .NET virtual machine – where security checks are inserted around calls to security-relevant methods. We prove that this algorithm achieves sound and complete mediation and transparency for a simplified model of MSIL, and we report on our experiences with the implementation of the algorithm for full MSIL.

Keywords: security policy enforcement, inline reference monitor.

1 Introduction

In today's networked world, code mobility is ubiquitous. Applications can be downloaded over the internet, or received as an attachment of emails. This support for applications from potentially untrustworthy sources comes with a serious risk: malicious or buggy applications can lead to denial of service, financial damage, leaking of confidential information and so forth.

One important class of countermeasures addresses this risk by monitoring the application at run time, and aborting it if it violates a predefined security policy. The events monitored are called *security-relevant events*, and they typically are operating system calls, or platform API method calls.

This paper is about such policy enforcement systems that are rich enough to enforce policies specified by means of a security automaton [1], an automaton that defines what sequences of security-relevant events are acceptable. Several such systems have been designed and prototyped [2,3,4], and security automata

* Dries Vanoverberghe is a research assistant of the Fund for Scientific Research - Flanders (FWO).

G. Barthe and F. de Boer (Eds.): FMOODS 2008, LNCS 5051, pp. 240–258, 2008.

have been shown to express exactly the policies enforceable by run-time monitoring [1]. The code access security architectures present in Java and .NET are an instance of such systems [2].

In standard policy enforcement systems, monitoring applications is integrated into the execution system or the trusted libraries. This makes it fairly easy to show complete mediation [5], the property that the monitor sees every occurrence of a security-relevant event. This paper zooms in on the inlined reference monitor (IRM) [6] approach, that rewrites untrusted applications and embeds the monitor directly into the application itself. Ensuring complete mediation is non-trivial when the program rewriter operates on an object-oriented intermediate language with state-of-the-art features such as virtual methods and delegates.

This paper proposes a caller-side rewriting algorithm for MSIL – the bytecode of the .NET virtual machine – where security checks are inserted around calls to security-relevant methods. We prove that this algorithm achieves sound and complete mediation and transparency (the property that the behavior of the rewritten program does not change if it is compliant with the enforced policy) for a simplified model of MSIL, and we report on our experiences with the implementation of the algorithm for full MSIL.

Section 2 elaborates on the problem statement. In Section 3, we introduce a simplified model of MSIL and the .NET Common Language Runtime (CLR) – the .NET virtual machine, and we propose a program rewriter that achieves complete mediation. Sections 4 and 5 explain how this system can be extended to support virtual method dispatch and delegates. In Section 6, we describe the implementation of our program rewriter for the full .NET CLR. Finally, we cover related work and conclude in Sections 7 and 8.

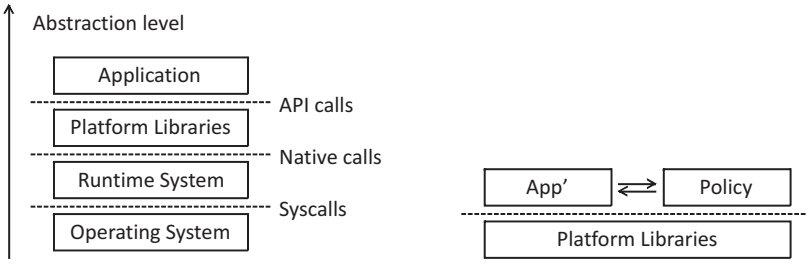
2 Problem Statement

The design space for IRM systems is rich, and different designs have different advantages and disadvantages. In particular, the problem of proving complete mediation is harder for some designs than for others. We discuss some of the design parameters, and motivate our design choices.

A first important design choice is the security-relevant events. Events can range from individual bytecode or machine instructions [7] over operating system calls [8] to Java method calls [6]. The trade-offs of these choices are discussed in [7], and broadly speaking the conclusion is that fine-grained monitoring allows expressive policies, but coarser-grained monitoring makes policies simpler to understand and write, and it is sufficiently expressive for practical purposes. Our system monitors method invocations.[1]

Furthermore, the abstraction level of the method calls monitored is important. Figure 1(a) shows a simplified architecture for a managed .NET application. The application calls methods of the API of the platform library, for instance a method to send data over a network socket. The implementation of this method

[1] A method invocation is when the execution enters a new method. Method calls are first dispatched to find the actual target method before they are invoked.



(a) Abstraction level (b) Rewritten application

```
public void ClientMethod(){
  //security checks
  Event();
  //security checks
}
```

(c) caller-side

```
public void Event(){
  //security checks
  //original code
  //security checks
}
```

(d) callee-side

Fig. 1. Design decisions for a policy enforcement systems

calls a lower-level native method, implemented in the runtime system. The native methods in turn performs system calls.

A second important design choice is whether to monitor the high-level methods that the application calls directly, or the low-level methods that are the most primitive abstraction of the system resources. Current research usually expresses security policies in terms of low-level methods [6,3]. This makes it easy to write policies that limit access to system resources, for instance limit the amount of network traffic, or file access. The higher level methods need not be monitored directly, because their implementations will call the lower level methods.

Unfortunately, as current system libraries can be complex, it can be hard for application developers to determine the security-relevant behavior of their application, since they make use of high-level methods. Furthermore, policy writers often want to selectively allow certain high-level methods, even if it executes a low-level method that is forbidden (e.g. using a logging method, even if access to the file system is not allowed).

In this paper, we use the high-level platform API methods to write policies. Security-relevant events are invocations of methods defined in the trusted system libraries from inside the untrusted application. This approach is similar to monitoring unmanaged applications at the boundary between kernelspace and userspace. In other words, we monitor the control flow transitions between Application and Platform Libraries in Figure 1(a).

A third design parameter is where to insert security checks. With *callee-side* program rewriting, the rewriter inserts checks inside the body of security-relevant methods (See Figure 1(d)). In a safe execution mechanism, it is fairly easy to show that untrusted applications can not circumvent security code. On the other hand, selectively allowing calls based on the call site is much harder. Since we

only want to monitor calls originating from the (untrusted) application, callee-side program rewriting is troublesome. Also, the rewriter needs to modify the trusted system libraries and this can be impossible, for instance when they are in ROM on a mobile device, or when a third party performs the inlining of the application as a service (as proposed in the S3MS project [9]).

In this paper, we use *caller-side* program rewriting, where the rewriter inserts checks at the call site (See Figure 1(c)).

Finally, a fourth design choice is what to inline: the policy enforcement code itself, or just a call to an external component that implements the policy. Both approaches are used in other systems, and the differences between the two approaches are minor. For convenience reasons – it is easier to formulate the complete mediation property we want to prove – we inline calls to a separate Policy Decision Point (PDP).

Figure 1(b) shows how the untrusted application is transformed into a new application and this new application invokes a method in the PDP before and after each security-relevant event.

Illustration of the Issues

IRM's are a powerful policy enforcement mechanism, but showing that a monitored application cannot subvert the security checks can be complex, in particular for IRM's based on caller-side rewriting. The security checks are inlined statically, and executed before the method call has been dispatched. In modern execution systems, this raises a number of important challenges:

- When using virtual methods or interfaces, the target method can usually not be determined statically. At runtime, the target is determined using the runtime type of the target object. An attacker could try to circumvent the security policy by casting an object to its base type and executing a security-relevant method using virtual dispatch. Alternatively, an attacker might inherit from the trusted system libraries or override the behavior of a security-relevant method.
- With delegates, the situation is even worse. A delegate points to a set of methods, and methods can be added to or removed from this set at runtime. Invoking the delegate invokes all methods in the set. An attacker could try to hide a call to a security-relevant method by adding it to a delegate and calling this delegate.

In short, achieving complete mediation with a caller-side program rewriter is much more challenging than using callee-side program rewriter. This is the key problem addressed in this paper.

3 Base System

In this section, we prove sound and complete mediation and transparency for a simplified model of the .NET CLR [10].

3.1 Execution System

Our formal model of the .NET CLR is based on Fruja's formalization [11]. Our model in this section supports assemblies (the .NET components, similar to Java's jar files), classes with (possibly static) non-virtual methods, and exceptions. Later sections discuss the extension to virtual methods. We do not model interfaces, value types or multithreading. We briefly describe the formal model, but since it is relatively straightforward, the reader is referred to [11] for details.

The execution system is a virtual machine that loads two assemblies: the untrusted application (U), and the trusted platform libraries (T). We assume that U has a method *main* without input arguments to start the execution. Each assembly contains a set of classes. A *type* is a pair consisting of an assembly and a class name defined in that assembly.

A *method reference* is a pair of a type and a method name (written as $Type :: MethodName$). The function *retType* maps a method reference to the type of the return value (or the special return type **void**). *locTypes* maps a method reference to a list of types for the local variables of that method. The function *argTypes* defines a list of types for the arguments of a method reference. If the method is an instance method, the first argument is the implicit this argument. Finally, the function *code* maps a method reference into a list of instructions, the bytecode representation of the body of the method. Table 1 summarizes the instructions that we will need in this paper. The virtual machine supports more instructions, but their semantics is straightforward.

An *exception handler* is a five-tuple defining the position of the beginning (inclusive) and end (exclusive) of the try block, the type of exception that is caught, and the position of the beginning and end of the catch block of the handler. The function *excHa* maps a method reference into a list of exception handlers.

The *execution state* of the virtual machine consists of a heap, a list of activation records and an exception record. The *heap* is modeled as a finite map from addresses to values. *heapLookup(heap, address)* looks up the value at a given address in the heap. Furthermore, *heapUpdate(heap, address, value)* returns a new heap that results from replacing the content of heap at the given address by the given value. An *activation record* is a five-tuple consisting of the current program counter, a list of addresses of the local variables, a list of addresses for the arguments, a stack of values (the evaluation stack) and the method reference of the method that is being executed. The *exception record* is a pair containing the currently active exception, and the next exception handler that will be tested to handle an exception. In our formalization, the active exception is always a value of the type *(T,Exception)* or *(T, SecurityException)*(where T is the trusted assembly).

Operational semantics
In a given state, *instr* denotes the current instruction that will be considered by the execution system. It is shorthand for $code(mref)[pc]$ with $mref$ the method

Table 1. Instructions

instruction	explanation
ldc.x	load a constant on the evaluation stack
ldloc i	load the value of the local variable at index i on the stack
stloc i	store the top value of the stack in the local variable at index i and pop it from the stack
brtrue k	branch to the instruction at index k if the top value of the stack is true and pop it from the stack
br k	unconditional branch to the instruction at index k
call *mref*	call the method mref using the arguments on the stack (and remove them from the stack) and put the return value on the stack (if return type not **void**)
ret	return from a method (and use the top value of the stack as return value if return type not **void**)
throw	throw the top value of the stack as an exception
newobj *mref*	create a new object using the constructor mref and put a reference to the new object on the stack

that is currently being executed (the top frame of the activation records), and *pc* the current program counter within this method.

An execution state is in *normal execution mode* if the active exception of the exception record is null. Table 1 informally explains the instructions that are relevant for our algorithm, and Appendix A defines the operational semantics for the instructions formally.

An execution state is in *exception handling mode* if it is not in normal execution mode. Appendix A also defines the operational semantics for exception handling. Exception handling mode is entered with the **throw** instruction, and it starts by looking for suitable exception handlers in the current method. An exception handler is suitable if the current program counter is inside the scope of the try block, and if the type of the exception is a subtype of the type of the exception handler. If a suitable handler is found, the evaluation stack is cleared and the active exception is pushed onto the stack. Execution continues in normal mode with the first instruction of the catch block of the exception handler.

If all exception handlers of the current method have been considered, the exception is propagated to the caller of the current method, and the execution mechanism continues searching at the first handler of the caller.

3.2 The Inlining Algorithm

Recall from Figure 1(b) that an inlining algorithm takes an untrusted assembly *App* as input, and it outputs a new assembly *App*′ that notifies the PDP of all security-relevant events. The security-relevant events are the entering into and exiting (both normally or exceptionally) from *security-relevant methods*. The security-relevant methods are a subset (designated by the policy writer) of the methods defined in the trusted assembly. Notification of the PDP is done by calling so-called *policy methods* in the PDP.

Definition 1 (Policy methods). *The functions $before$, $after$ and $excep$ map a security-relevant method $mref$ onto the corresponding policy methods $before(mref)$, $after(mref)$ and $excep(mref)$ that are called respectively before entering $mref$, after successful return, and after exceptional return. $before(mref)$ has the same argument types as $mref$, $after(mref)$ has an additional parameter for the return value, and $excep(mref)$ has an additional parameter for the exception.*

In our simplified model, we require policy methods to be part of the trusted assembly. They throw a SecurityException when the policy is violated, and return **void** otherwise. Policy methods are not allowed to call back into the untrusted assembly. The untrusted application (before inlining) should not contain any calls to policy methods.

Our inlining algorithm is defined in Figure 2. The algorithm goes through the instruction list of all methods in the untrusted assembly, looking for **call** instructions to a security-relevant method. Each such call instruction is replaced by a block of code generated using the *generateCode* function, defined below. We say that such call instruction has been *processed* by the inliner. For simplification purposes, we do not intercept constructor calls, but their treatment is identical to a call to a static method with the new object as return value.

Definition 2 (generateCode). *Given an instruction $instr$ of the form* **call** *$mref$, a list of types $localvars$, a list of exception handlers $excha$ and the index of the instruction i, the function $generateCode(instr, localvars, excha, i)$ returns*

$Body = (List(Instr), List(Type), List(ExceptionHandler))$

$inlinebody : Map(Body, Body)$
$inlinebody((body, localvars, excha)) = inlinebodyhelp([], body, 0, localvars, excha)$

$inlinebodyhelp(left, [], \_, localvars, excha) =$
  return $(left, localvars, excha)$
$inlinebodyhelp(left, right, i, localvars, excha) =$
  $right == instr, right'$
  if $(instr == call\ mref\ and\ mref\ is\ a\ SRM)$ {
    $(newCode, localvars', newExcha) = generateCode(instr, localvars, excha, i)$
    $n = \#newCode$
    $left' = patchBranchTargets(left, n, i)$
    $right'' = patchBranchTargets(right', n, i)$
    $left'' = left', newCode$
    $excha' = newExcha, patchExchaPcs(excha, n, i)$
    return $inlinebodyhelp(left'', right'', i + n, localvars', excha')$
  } **else** {
    $left' = left, instr$
    return $inlinebodyhelp(left', right', i + 1, localvars, excha)$
  }

Fig. 2. Program rewriter algorithm

```
        stloc m + n
        ...                          1. Store arguments
        stloc m
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        ldloc m
        ...                          2. Reload arguments
        ldloc m + n
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        call before(mref)            3. Call before method
    try{
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        ldloc m
        ...                          4. Reload arguments
        ldloc m + n
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        call mref                    5. Call original method
        stloc m + n + 1              6. Store return value
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        ldloc m
        ...                          7. Reload arguments and return value
        ldloc m + n + 1
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        call after(mref)             8. Call after method
        ldloc m + n + 1              9. Reload return value
        br end                       10. goto end
    } catch (SecurityException) {
        throw                        11. Rethrow exception
    } catch (Exception) {
        stloc m + n + 2              12. Store exception
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        ldloc m
        ...                          13. Reload arguments and exception
        ldloc m + n
        ldloc m + n + 2
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        call exception(mref)         14. Call exception method
        ldloc m + n + 2              15. Reload exception
        throw                        16. Rethrow exception
end: }
```

Fig. 3. Generated code fragment where n is the number of arguments of the target method, and m is the number of pre-existing local variables in the method that is being rewritten.

a block of code defined by the template in Figure 3, a modified list of types for local variables and a modified list of exception handlers.

The insertion of the code fragments generated by *generateCode* changes the locations of the original instructions in the method being rewritten. Since these locations are used in branch instructions and in exception handlers, we need to patch these. So the inlining algorithm patches branch targets and exception handlers using the function $patchLocation(n, i) = \lambda loc.\textbf{if}\ (loc > i)\ \textbf{then}\ loc + n - 1\ \textbf{else}\ loc$, where i is the location of the call being processed, and n is the size of the generated code block being inserted.

We discuss in more detail the block of code generated by *generateCode* (Figure 3). First, we generate *stloc* instructions to store the arguments that are on the evaluation stack into new local variables. Because these variables are new, and this is the only place in the generated code where values are stored into these variables, their value does not change after this point. This ensures that each time the variables are loaded on the stack, the relevant values on the top of the stack are the same values as the values initially on the stack.

Next, we use *ldloc* instructions to load the arguments on the stack again, and a *call* instruction to invoke the before method of the policy.

The remaining part of the code is a try catch structure. In the try scope, we first generate instructions to load the arguments on the stack and call the original security-relevant method. Then we store the return value in a local variable (if the return type is not void). For the after method of the policy, we generate instructions to load the arguments and the saved return value on the stack and to call the after method. Finally, we load the return value on the stack again, and we branch out of the try block.

The type of the first exception handler is *SecurityException*. Because we assume that security-relevant methods do not throw security exceptions, this exception comes from the after method. We generate code to simply rethrow the original exception.

The second exception handler catches any type of exceptions. Because we assume that the policy methods are total, this exception is raised inside the security-relevant method and the exceptional method of the policy must be executed. We generate instructions to store the exception into a local variable, reload the arguments, reload the exception, call the exceptional method of the policy and finally reload and rethrow the exception.

3.3 Properties of the Inlining Algorithm

The first property we consider is *complete mediation* [5]: every security-relevant event is seen by the monitor.

We say that a method invocation, return or exceptional return is *observable* when it crosses the boundary between the untrusted assembly and the trusted assembly. The function $observable(mref, mref')$ returns true if one of its arguments is defined in the trusted assembly, and the other one in the untrusted assembly.

An abstract trace of a program is a (possibly infinite) list of observable method invocations, returns and exceptional returns.

Definition 3 (Abstract trace). *The abstract trace of a program is the list of observable method invocations, returns and exceptional returns that occur when executing the main method of the program.*

Figure 4 shows how to compute the abstract trace, based on the operational semantics.

For simplification purposes, we assume that security-relevant methods do not call back into untrusted applications and they terminate. Furthermore, we

$$\frac{\begin{array}{c}(H,S,E) \to (H',S',E') \\ \#S' = \#S + 1 \qquad S' = \_,(pc,\_,argAdr,\_,mref) \qquad S = \_,(\_,\_,\_,\_,oldmref) \\ code(oldmref)[pc] \neq newobj\ \_ \qquad observable(mref,oldmref) \\ argTypes(mref) == \overrightarrow{A}_{1..n} \qquad \overrightarrow{v} = heapLookup(H',argAdr)\end{array}}{trace(H,S,E) = Enter(mref,\overrightarrow{v}), trace(H',S',E')}$$

$$\frac{\begin{array}{c}(H,S,E) \to (H',S',E') \qquad \#S' = \#S - 1 \qquad S = \_,(pc,\_,\_,evalStack,mref) \\ S' = \_,(\_,\_,\_,\_,oldmref) \qquad observable(mref,oldmref) \qquad E = (null,\_) \\ code(mref)[pc] = ret \qquad retType(mref)! = void \qquad evalStack = evalStack',v\end{array}}{trace(H,S,E) = Return(v), trace(H',S',E')}$$

$$\frac{\begin{array}{c}(H,S,E) \to (H',S',E') \qquad \#S' = \#S - 1 \\ S = \_,(pc,\_,\_,\_,mref) \qquad S' = \_,(\_,\_,\_,\_,oldmref) \qquad observable(mref,oldmref) \\ E = (null,\_) \qquad code(mref)[pc] = ret \qquad retType(mref) == void\end{array}}{trace(H,S,E) = Return, trace(H',S',E')}$$

$$\frac{\begin{array}{c}(H,S,E) \to (H',S',E') \qquad \#S' = \#S - 1 \qquad S = \_,(\_,\_,\_,\_,mref) \\ S' = \_,(\_,\_,\_,\_,oldmref) \qquad observable(mref,oldmref) \qquad E = (v,\_) \qquad v! = null\end{array}}{trace(H,S,E) = Exception(v), trace(H',S',E')}$$

$$\frac{(H,S,E) \to (H',S',E') \qquad \#S' = \#S}{trace(H,S,E) = trace(H',S',E')}$$

Fig. 4. Computation of the trace of a program

assume that security-relevant methods always return a value to keep the properties simple.

Definition 4 (Complete mediation). *An abstract execution trace of an untrusted assembly program satisfies complete mediation if and only if for each index i in trace such that trace[i] = Enter(mref, vals) and mref is security-relevant:*

- *trace[i-2]=Enter(before(mref),vals) and trace[i-1]=Return*
- *trace[i+1]=Return(val) implies trace[i+2]=Enter(after(mref),vals . val)*
- *trace[i+1]=Exception(val) implies trace[i+2]=Enter(excep(mref),vals . val)*

We prove that our inliner always produces programs with completely mediated traces. First we need some technical lemmas.

Lemma 1 (No jumps into inlined code). *For every untrusted assembly P, let P′ = inline(P). For all possible runs of P′, control flow can enter a code block inserted during inlining only through the first instruction of the block.*

Proof. The patching of the branch targets makes sure that there are no jumps into the generated code, thus control flow must enter through the first instruction.

Lemma 2 (Complete replacement). *Any call instruction that invokes a security-relevant method at run time, has been processed by the inliner.*

The proof of this lemma is trivial in this execution system. However, this is the key lemma that breaks when adding delegates or virtual method dispatch. Later sections discuss how to maintain this lemma in the presence of delegates and virtual calls, and this will require improvements to the inliner.

Lemma 3 (Abstract trace of generated code). *The code blocks output by generateCode can only generate the following traces:*

1. *Enter(before(mref), vals), Exception(exc),... with exc a SecurityException.*
2. *Enter(before(mref), vals), Return, Enter(mref, vals), Exception(exc), Enter(excep(mref), vals . exc), Exception(exc2), ... with exc not a SecurityException and exc2 a SecurityException.*
3. *Enter(before(mref), vals), Return, Enter(mref, vals), Exception(exc), Enter(excep(mref), vals . exc), Return, ... with exc not a SecurityException.*
4. *Enter(before(mref), vals), Return, Enter(mref, vals), Return(val), Enter(after(mref), vals . val), Exception(exc), ... with exc a SecurityException.*
5. *Enter(before(mref), vals), Return, Enter(mref, vals), Return(val), Enter(after(mref), vals . val), Return, ...*

Proof. Taking into account the restrictions that (1) policy methods do not call back into the untrusted assembly, and (2) policy methods can only throw SecurityExceptions or return void, the proof is straightforward.

Theorem 1 (Complete mediation of the algorithm). *For every untrusted assembly P, let $P' = inline(P)$. The abstract trace $trace'$ of P' satisfies complete mediation.*

Proof. For each $Enter(mref, vals)$ in $trace'$ where $mref$ is security-relevant :

1. Using the computation rules for traces (Figure 4), the Enter can only be caused by a call instruction.
2. By Lemma 2, each call instruction that can potentially result in invoking a security-relevant method has been replaced by a block of generated code.
3. Using Lemma 1, the only way to start the execution of the generated block of code is by starting at the first instruction.
4. According to Lemma 3, the generated code can only lead to five possible traces. Because we have an $Enter(mref, vals)$ in the trace, trace 1 is not possible. In all other cases complete mediation holds by Definition 4.

The second property we consider is sound mediation.

Definition 5 (Sound mediation). *An abstract execution trace of an untrusted assembly program satisfies sound mediation if and only if for each index i in trace:*

- *trace[i]=Enter(before(mref),vals) and trace[i+1]=Return implies trace[i+2]=Enter(mref, vals)*

- *trace[i]=Enter(after(mref),vals . val) implies trace[i-2]=Enter(mref, vals) and trace[i-1]=Return(val)*
- *trace[i]=Enter(excep(mref),vals . val) implies trace[i-2]=Enter(mref, vals) and trace[i-1]=Exception(val)*

Theorem 2 (Sound mediation of the algorithm). *For every untrusted assembly P, let $P' = inline(P)$. The execution trace$'$ of P' satisfies sound mediation.*

Proof. Assuming that P did not contain any calls to the policy before rewriting, all calls to the policy in P' come from inside the generated code block. Using the generated traces (Lemma 3), completing the proof is trivial.

The third property we consider is *transparency*: a policy enforcement system is transparent if it has no observable effect on programs that satisfy the policy. We formalize this by considering the effect when inlining calls to policy methods that do nothing, they return immediately. We call such methods *passive*.

Definition 6 (Equality modulo policy calls). *Two abstract execution traces trace and trace$'$ are equal modulo policy calls if and only if they are equal after removing all calls to and returns from policy methods.*

Definition 7 (Transparency). *Let I be an inlining algorithm. Let P be a program, and let P' be the inlining of P with I using only passive policy methods. I is* transparent *if, for all such P, the trace of P' is equal modulo policy calls to the trace of P.*

Clearly, if policy methods are non-passive, a program may behave differently after inlining. In general, the desirable situation is that policy methods are indistinguishable from passive policy methods *until the untrusted program violates the policy.*

Theorem 3 (Transparency of the algorithms). *The inlining algorithm in Figure 2 is transparent.*

The proof of this theorem, an induction over evaluation steps of the original program, can be found in an extended version of this paper, published as a technical report [12]. The key observation is that the code blocks inserted by the inliner (Figure 3) have no observable effect on the traces of the program if the policy methods are passive. First, there is a strong relationship between the static structure – instructions, exception handlers, and variables of method bodies – of P and P'. Over this structural relationship, we define the notion of structural equivalence of an execution state of P and P'. We prove that for each execution state reachable in k steps in P, there exists an execution state that is reachable in $l \geq k$ steps in P' that is structurally equivalent and the traces to reach these states are equal modulo calls to the policy.

4 Virtual Methods

To support virtual method calls in our model of the .NET CLR, the following extensions are needed. The VM keeps track of a map of object references to their actual type (the function $actualTypeOf$). The new instruction *callvirt* performs a virtual call: it looks up the method to be invoked dynamically based on the run-time type of the receiver, and then behaves like a regular call. Figure 11 in the appendix gives the evaluation rule for the *callvirt* instruction.

If our inliner would treat *callvirt* in the same ways as *call*, complete mediation would break. There are three issues to be dealt with.

First, lemma 2 would break. Suppose a security-relevant method $B :: m$ overrides a non-security-relevant method $A :: m$ in a superclass A (Figure 5). A virtual call with static target $A :: m$ would not be processed by the inliner, yet it could lead to an invocation of $B :: m$ at run time.

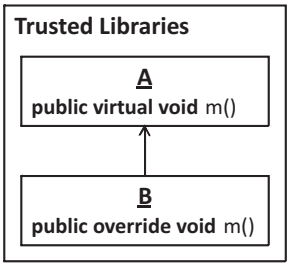


Fig. 5. Violating complete replacement

We deal with this first issue by requiring that the set of security-relevant methods is upward closed.

Definition 8 (Upwards closure of a set of methods). *A set of methods is upwards closed if and only if no method in the set overrides a method that is not in the set.*

In practice, this is not a real restriction, as methods can always be added to the set of security-relevant methods, even if the policy does impose constraints on them.

The second issue arises when inheriting security-relevant methods. Suppose $A :: m$ is a security-relevant method, and the untrusted assembly creates a subclass C of A that does not redefine m (Figure 6). Then a call that statically looks like $C :: m$ is actually a call to $A :: m$. Clearly, we can not require $C :: m$ to be in the set of security-relevant methods, as it is in the untrusted application, and the policy writer typically does not even know it exists.

We address this issue by forbidding inheritance of security-relevant methods in the untrusted assembly. Again, this is no restriction, as overriding is allowed, and the untrusted application could override the security-relevant method and

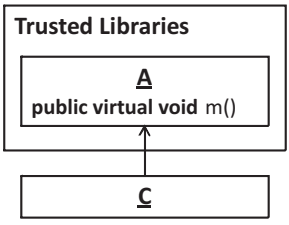


Fig. 6. Inheriting security-relevant methods

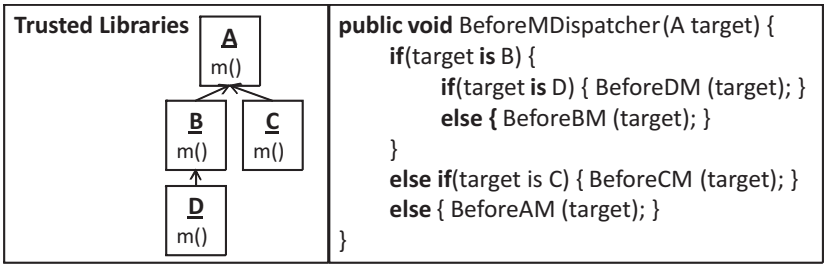


Fig. 7. Dispatching methods

then simply do a base call if the same behavior as the original method is desired. In fact, such a transformation could even be done automatically when no redefinition is found. The base call in the transformed program will be recognized by the inliner as a statically bound call to a security-relevant method.

The third and last issue to address is how to deal with dynamic dispatch to policy methods. Since the method called is determined dynamically, the determination of the appropriate policy methods now also needs to be done dynamically. We handle this by creating a dispatching method for each virtual method that is security-relevant (See Figure 7). The dispatching method uses runtime tests on the actual type of the target to determine the actual method that will be invoked. If this target method is security-relevant, then the dispatching method calls the corresponding policy method. Otherwise, it returns from the dispatching method. The program rewriter now inserts calls to these dispatching methods instead of the actual policy methods.

With these three extensions, the resulting inliner is completely and soundly mediating and transparent in the presence of virtual calls.

5 Delegates

Delegates are essentially type safe function pointers. A delegate encapsulates a set of method references of the same signature as the delegate. Calling the delegate invokes all the methods in the set. A delegate can be passed on to other

code where it is possible to call the delegate without statically knowing its target methods. Therefore delegates are challenging for a caller-side inliner.

To maintain the complete mediation property of our inliner, we enforce the property that the untrusted application never holds a reference to a delegate that contains a pointer to a security-relevant method. By consequence, a call to a delegate is never a call to a security-relevant method.

First, we extend the program rewriter to deal with the case where the untrusted assembly creates delegates. To create a delegate containing a method pointer, an application must first load the method pointer on the stack (using the *ldftn* or *ldvirtftn* instruction), and then it must call a Delegate subclass constructor. When the program rewriter parses an attempt to load a security-relevant method pointer on the stack, it creates a new wrapper method inside the application that does a normal call to the security-relevant method, and a pointer to this wrapper method is pushed on the stack. Hence the delegate will not point to the security-relevant method, but to the wrapper method. The program rewriter transforms the body of the wrapper method to insert calls to the policy, thus preserving complete mediation.

Second we need to impose a constraint on the trusted libraries: no method in the trusted API should create delegates that contain security-relevant methods and pass them to the untrusted assembly in any way. For the security-relevant methods we have considered so far, the trusted libraries never create delegates that contain security-relevant methods.

With these extensions, our program rewriter is soundly and completely mediating and transparent, even in the presence of delegates.

6 Implementation

The research reported on in this paper is done in the context of the project Security of Software and Services for Mobile Systems (S3MS) [9]. The inlining algorithm described in this paper has been implemented for the full .NET Framework, and for the .NET Compact Framework running on mobile devices such as smartphones. We used the Mono.Cecil [13] libraries to parse and generate MSIL.

The execution system we describe in this paper hides a part of the complexity of the real .NET CLR. While implementing our approach, we encountered some challenging issues (in addition to the ones we already discussed). For example, the real CLR forbids entering a protected region (a try block) with a non-empty evaluation stack and it resets the evaluation stack when leaving a protected region. Our approach inserts new try-catch handlers, thus we store the entire evaluation stack before entering the try block, instead of just saving the arguments for the security-relevant method. To do so, we implemented a simple one-pass data flow analysis to compute the types on the stack. We store the values on the stack in local variables, and reload them after the catch-blocks.

Furthermore, the real CLR forbids entering a protected region when the current object is not fully initialized in constructors. Initialization statements for fields are executed before an object is fully initialized. If these statements

use security-relevant methods, the program rewriter inserts try-catch handlers and the current object is not fully initialized upon entering the try block. We solved this issue in our implementation by generating wrapper methods for these security-relevant methods (like we did with delegates).

Our experience with the implementation is promising. The performance overhead of the inlining is very small (this confirms performance measurements done for other IRM's [2]), and the inliner can handle real-world applications that are used as case studies in the S3MS project.

7 Related Work

Both the current Java Virtual Machine [14] and Microsofts Common Language Runtime [10] use stack inspection to enforce security policies on untrusted code. The security policies that can be enforced in our approach are more flexible, since the entire history of events can be used. Resource constraints are an example of the kind of policies that are not enforceable using stack inspection.

SASI [7] and PoET/PSLang [6] are two policy enforcement tools based on security automata [1]. Both techniques are based on inline reference monitors. Sasi's event system is very powerfull, as arbitrary machine instructions can be monitored, but it has a higher performance impact. PoET/PSLang targets Java applications and Erlingsson and Schneider have shown that it can be used to enforce stack inspection based policies, and that the performance is competitive [2]. In contrast with our approach, they make no claims about complete mediation or transparency.

Naccio [4] also monitors security-sensitive operations, but instead of inserting instructions inside the application or the system libraries, a new wrapper library is created that enforces the security policy and delegates control back to the original libraries.

Polymer [3] is a policy enforcement system based on edit automata [15]. To master the complexity of policies, Polymer supports composition of complex security policies using smaller building blocks. In contrast with our approach, polymer uses callee-side rewriting to enforce security policies.

The work in this paper can been seen as a particular instantiation of Aspect Oriented Programming [16] targeting security policy enforcement on untrusted code. Because our rewriting algorithm is much simpler than the weaving mechanisms in full AOP, it is easier to prove correctness.

The abstract traces in our system are similar to the fully abstract trace semantics of JavaJr [17].

8 Conclusions

In this paper, we propose a caller-side rewriting algorithm for MSIL – the bytecode of the .NET virtual machine – where security checks are inserted around calls to security-relevant methods.

The algorithm has been implemented and can deal with real-world .NET applications. Moreover, the algorithm has been proven correct for a simplified model of MSIL and the .NET virtual machine. To the best of our knowledge, this is the first provably correct inlining algorithm for an intermediate language that supports both virtual methods and delegates.

References

1. Schneider, F.B.: Enforceable security policies. ACM Trans. on Information and System Security 3(1), 30–50 (2000)
2. Erlingsson, U., Schneider, F.B.: IRM enforcement of Java stack inspection. In: IEEE Symposium on Security and Privacy, pp. 246–255 (2000)
3. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with polymer. In: PLDI 2005, pp. 305–314. ACM Press, New York (2005)
4. Evans, D., Twyman, A.: Flexible policy-directed code safety. In: IEEE Symposium on Security and Privacy, pp. 32–45 (1999)
5. Saltzer, J., Schroeder, M.: The protection of information in computer systems. IEEE 9(63) (1975)
6. Erlingsson, U.: The inlined reference monitor approach to security policy enforcement. PhD thesis, Cornell University (2004); Adviser-Fred B. Schneider
7. Erlingsson, S.: SASI enforcement of security policies: A retrospective. In: WNSP: New Security Paradigms Workshop. ACM Press, New York (2000)
8. Provos, N.: Improving host security with system call policies. In: SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium, p. 18. USENIX Association, Berkeley, CA, USA (2003)
9. S3MS: Security of software and services for mobile systems (2007), `http://www.s3ms.org/`
10. European Computer Machinery Association: Standard ECMA-335: Common Language Infrastructure, 4th edn. (June 2006)
11. Fruja, N.G.: Type Safety of C# and.NET CLR. PhD thesis, ETH Zurich (2006)
12. Vanoverberghe, D., Piessens, F.: A caller-side inline reference monitor for object-oriented intermediate language: Extended version (2008), `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW512.abs.html`
13. Evain, J.: Cecil, `http://www.mono-project.com/Cecil`
14. Lindholm, T., Yellin, F.: The Java(TM) Virtual Machine Specification, April 1999. Prentice Hall PTR, Englewood Cliffs (1999)
15. Ligatti, J., Bauer, L., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. International Journal of Information Security 4(1–2), 2–16 (2005)
16. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
17. Jeffrey, A.S.A., Rathke, J.: Java jr.: Fully abstract trace semantics for a core Java language. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 423–438. Springer, Heidelberg (2005)

Appendix

A Operational Semantics

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \qquad instr = \textbf{ldc.x}\ v \qquad evalStack' = evalStack, v \qquad S'' = S', (pc+1, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H, S'', E)}$$

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \qquad instr = \textbf{ldloc}\ i \qquad adr = locAdr[i] \qquad v = heapLookup(H, adr) \qquad evalStack' = evalStack, v \qquad S'' = S', (pc+1, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H, S'', E)}$$

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \qquad instr = \textbf{stloc}\ i \qquad evalStack = evalStack', v \qquad adr = locAdr[i] \qquad H' = heapUpdate(H, adr, v) \qquad S'' = S', (pc+1, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H', S'', E)}$$

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \qquad instr = \textbf{ldarg}\ i \qquad adr = argAdr[i] \qquad v = heapLookup(H, adr) \qquad evalStack' = evalStack, v \qquad S'' = S', (pc+1, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H, S'', E)}$$

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \qquad instr = \textbf{starg}\ i \qquad evalStack = evalStack', v \qquad adr = argAdr[i] \qquad H' = heapUpdate(H, adr, v) \qquad S'' = S', (pc+1, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H', S'', E)}$$

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \qquad instr = \textbf{br}\ target \qquad S'' = S', (target, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H, S'', E)}$$

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \qquad instr = \textbf{brtrue}\ target \qquad evalStack = evalStack', v \qquad v == 0 \qquad S'' = S', (pc+1, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H, S'', E)}$$

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \qquad instr = \textbf{brtrue}\ target \qquad evalStack = evalStack', v \qquad v! = 0 \qquad S'' = S', (target, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H, S'', E)}$$

Fig. 8. Evaluation rules for normal execution (Part 1)

$$\frac{\begin{array}{c}S = S', (pc, locAdr, argAdr, evalStack, mref) \\ instr = \textbf{call}\ T :: M \qquad argTypes(T :: M) = \overrightarrow{A}_{1..n} \\ evalStack = evalStack', \overrightarrow{v}_{1..n} \qquad argAdr' = \overrightarrow{a}_{1..n}\ (with\ a_i\ fresh) \\ H' = heapUpdate(H, \overrightarrow{a}_{1..n}, \overrightarrow{v}_{1..n}) \qquad locTypes(T :: M) = \overrightarrow{L}_{1..m} \\ locAdr' = \overrightarrow{l}_{1..m}\ (with\ l_i\ fresh) \qquad H'' = heapUpdate(H', \overrightarrow{l}_{1..m}, defVal(\overrightarrow{L}_{1..m})) \\ S'' = S', (pc, locAdr, argAdr, evalStack', mref), (0, locAdr', argAdr', [], T :: M)\end{array}}{(H, S, E) \rightarrow (H'', S'', E)}$$

$$\frac{\begin{array}{c}S = S', (pc2, locAdr2, argAdr2, evalStack2, mref2), (pc, locAdr, argAdr, evalStack, mref) \\ instr = \textbf{ret} \\ retType(mref) = void \qquad S'' = S', (pc2 + 1, locAdr2, argAdr2, evalStack2, mref2)\end{array}}{(H, S, E) \rightarrow (H, S'', E)}$$

$$\frac{\begin{array}{c}S = S', (pc2, locAdr2, argAdr2, evalStack2, mref2), (pc, locAdr, argAdr, evalStack, mref) \\ instr = \textbf{ret} \qquad retType(mref) \neq void \qquad evalStack = evalStack', v \\ evalStack2' = evalStack2, v \qquad S'' = S', (pc2 + 1, locAdr2, argAdr2, evalStack2', mref2)\end{array}}{(H, S, E) \rightarrow (H, S'', E)}$$

$$\frac{\begin{array}{c}S = S', (pc, locAdr, argAdr, evalStack, mref) \\ instr = \textbf{throw} \qquad evalStack = evalStack', val \\ E' = (val, 0) \qquad cal \neq null \qquad S'' = S', (pc, locAdr, argAdr, evalStack', mref)\end{array}}{(H, S, E) \rightarrow (H, S'', E')}$$

Fig. 9. Evaluation rules for normal execution (Part 2)

$$\frac{\begin{array}{c}S = S', (pc, locAdr, argAdr, evalStack, mref) \\ E = (val, n) \qquad n = \#excHa(mref) \qquad E' = (val, 0)\end{array}}{(H, S, E) \rightarrow (H, S', E')}$$

$$\frac{\begin{array}{c}S = S', (pc, locAdr, argAdr, evalStack, mref) \\ E = (val, n) \qquad n < \#excHa(mref) \qquad excHa(mref)[n] = (from, to, type, cfrom, cto) \\ from > pc\ \vee\ pc \geq to\ \vee\ actualTypeOf(val) \not< type \qquad E' = (val, n + 1)\end{array}}{(H, S, E) \rightarrow (H, S, E')}$$

$$\frac{\begin{array}{c}S = S', (pc, locAdr, argAdr, evalStack, mref) \\ E = (val, n) \qquad n < \#excHa(mref) \qquad excHa(mref)[n] = (from, to, type, cfrom, cto) \\ from \leq pc < to \qquad actualTypeOf(val) < type \\ E' = (null, 0) \qquad evalStack' = val \qquad S'' = S', (cfrom, locAdr, argAdr, evalStack', mref)\end{array}}{(H, S, E) \rightarrow (H, S'', E')}$$

Fig. 10. Evaluation rules for exception handling

$$\frac{\begin{array}{c}S = S', (pc, locAdr, argAdr, evalStack, mref) \\ instr = \textbf{callvirt}\ T :: M \qquad isVirtual(T :: M) = true \\ argTypes(T :: M) = T, \overrightarrow{A}_{1..n} \qquad evalStack = evalStack', t, \overrightarrow{v}_{1..n} \qquad \overrightarrow{args} = t, \overrightarrow{v}_{1..n} \\ T' :: M = lookupMethod(actualTypeOf(t), T :: M) \qquad argAdr' = \overrightarrow{a}_{0..n}\ (with\ a_i\ fresh) \\ H' = heapUpdate(H, \overrightarrow{a}_{1..n}, \overrightarrow{args}) \qquad locTypes(T' :: M) = \overrightarrow{L}_{1..m} \\ locAdr' = \overrightarrow{l}_{1..m}\ (with\ l_i\ fresh) \qquad H'' = heapUpdate(H', \overrightarrow{l}_{1..m}, defVal(\overrightarrow{L}_{1..m})) \\ S'' = S', (pc, locAdr, argAdr, evalStack', mref), (0, locAdr', argAdr', [], T' :: M)\end{array}}{(H, S, E) \rightarrow (H'', S'', E)}$$

Fig. 11. Evaluation rule for virtual calls

Author Index