Dirk Beyer
Michele Boreale (Eds.)

LNCS 7892

Formal Techniques for Distributed Systems

Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013
Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013
Florence, Italy, June 2013, Proceedings

ifip

Springer

Dirk Beyer Michele Boreale (Eds.)

Formal Techniques for Distributed Systems

Joint IFIP WG 6.1 International Conference
FMOODS/FORTE 2013
Held as Part of the 8th International Federated Conference
on Distributed Computing Techniques, DisCoTec 2013
Florence, Italy, June 3-5, 2013, Proceedings

Springer

Volume Editors

Dirk Beyer
University of Passau
Department of Computer Science and Mathematics
Innstraße 31, 94032, Passau, Germany

Michele Boreale
Università di Firenze
Dipartimento di Statistica, Informatica, Applicazioni (DiSIA)
Viale Morgagni, 65, 50134 Florence, Italy

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-38591-9 e-ISBN 978-3-642-38592-6
DOI 10.1007/978-3-642-38592-6
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013938531

CR Subject Classification (1998): F.3, D.2.4, F.1, D.2, F.4, I.2.2-3, D.3, C.2, C.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Foreword

In 2013, the 8th International Federated Conference on Distributed Computing Techniques (DisCoTec) took place in Florence, Italy, during June 3–6. They were hosted and organized by the Università di Firenze. The DisCoTec series of federated conferences, one of the major events sponsored by the International Federation for Information processing (IFIP), included three conferences:

- The 15th International Conference on Coordination Models and Languages (Coordination)
- The 13th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)
- The 2013 IFIP Joint International Conference on Formal Techniques for Distributed Systems (33rd FORTE/15th FMOODS)

Together, these conferences cover the complete spectrum of distributed computing subjects ranging from theoretical foundations to formal specification techniques to systems research issues.

Each of the first three days of the federated event began with a plenary speaker nominated by one of the conferences. The three invited speakers were: Tevfik Bultan, Department of Computer Science at the University of California, Santa Barbara, USA; Gian Pietro Picco, Department of Information Engineering and Computer Science at the University of Trento, Italy; and Roberto Baldoni, Department of Computer, Control and Management Engineering "Antonio Ruberti", Università degli Studi di Roma "La Sapienza", Italy. In addition, on the second day, there was a joint technical session consisting of one paper from each of the conferences. There were also three satellite events:

1. The 4th International Workshop on Interactions Between Computer Science and Biology (CS2BIO) with keynote talks by Giuseppe Longo (ENS Paris, France) and Mario Rasetti (ISI Foundation, Italy)
2. The 6th Workshop on Interaction and Concurrency Experience (ICE) with keynote lectures by Davide Sangiorgi (Università di Bologna, Italy) and Damien Pous (ENS Lyon, France)
3. The 9th International Workshop on Automated Specification and Verification of Web Systems (WWV) with keynote talks by Gerhard Friedrich (Universität Klagenfurt, Austria) and François Taïani (Université de Rennes 1, France)

I believe that this program offered each participant an interesting and stimulating event. I would like to thank the Program Committee Chairs of each conference and workshop for their effort. Moreover, organizing DisCoTec 2013

was only possible thanks to the dedicated work of the Publicity Chair Francesco Tiezzi (IMT Lucca, Italy), the Workshop Chair Rosario Pugliese (Università di Firenze, Italy), and the members of the Organizing Committee from Università di Firenze: Luca Cesari, Andrea Margheri, Massimiliano Masi, Simona Rinaldi, and Betti Venneri. To conclude I want to thank the International Federation for Information Processing (IFIP) and Università di Firenze for their sponsorship.

June 2013 Michele Loreti

Preface

This volume contains the proceedings of the 2013 IFIP Joint International Conference on Formal Techniques for Distributed Systems (33^{rd} FORTE/15^{th} FMOODS). The joint conference is the result of merging the two international conferences Formal Techniques for Networked and Distributed Systems (FORTE) and Formal Methods for Open Object-Based Distributed Systems (FMOODS). The city of Florence, Italy, was selected as the conference venue, taking place during June 3–5, 2013. This edition of the conference was organized as part of the 8th International Federated Conference on Distributed Computing Techniques (DisCoTec).

The FORTE/FMOODS conference series represents a forum for fundamental research on theory, models, tools, and applications for distributed systems. The conference encourages contributions that combine theory and practice, and that exploit formal methods and theoretical foundations to present novel solutions to problems arising from the development of distributed systems. FORTE/FMOODS covers distributed computing models and formal specification, testing, and verification methods. The application domains include all kinds of application-level distributed systems, telecommunication services, Internet, embedded and real-time systems, as well as networking and communication security and reliability.

We received a total of 49 full paper submissions for review (10 were withdrawn before review). Each submission was reviewed by at least three members of the Program Committee (papers that were co-authored by a PC member received four reviews). Based on high-quality reviews, and a thorough (electronic) discussion by the Program Committee, we selected 20 papers for presentation at the conference and for publication in this volume.

Tevfik Bultan, University of California, Santa Barbara (USA), was the keynote speaker of FORTE/FMOODS 2013. He is well-known in our community for his work on dependability of Web-service-based systems and their automated verification. Tevfik Bultan's keynote, entitled “Analyzing Interactions of Asynchronously Communicating Software Components,” gave an overview of “choreography” specifications and their realizability; an abstract of the keynote is included in this proceedings volume.

We would like to thank all who contributed to making FORTE/FMOOD 2013 a successful event: first of all, the authors, for submitting their fine research results; the Program Committee, for an efficient discussion and a fair selection process; the invited speaker; and of course the attendees of FORTE/FMOODS 2013! We are also grateful to the DisCoTec general chair, Michele Loreti, and all members of his local-organization team at the Università di Firenze. Thank you!

June 2013

Dirk Beyer
Michele Boreale

Organization

Program Committee

Sven Apel	University of Passau, Germany
Saddek Bensalem	VERIMAG, France
Dirk Beyer	University of Passau, Germany
Michele Boreale	Università di Firenze, Italy
Tevfik Bultan	University of California at Santa Barbara, USA
Luis Caires	Universidade Nova de Lisboa, Portugal
Mariangiola Dezani-Ciancaglini	Università di Torino, Italy
Juergen Dingel	Queen's University, Canada
Simon Gay	University of Glasgow, UK
Holger Giese	University of Potsdam, Germany
Kim Guldstrand Larsen	Aalborg University, Denmark
Arie Gurfinkel	Software Engineering Institute, USA
Matthew Hennessy	Trinity College Dublin, Ireland
Paola Inverardi	Università dell'Aquila, Italy
Alan Jeffrey	Bell Labs, USA
Joost-Pieter Katoen	RWTH Aachen University, Germany
Vladimir Klebanov	Karlsruhe Institute of Technology, Germany
Axel Legay	IRISA/INRIA at Rennes, France
Matteo Maffei	Saarland University, Germany
Uwe Nestmann	TU Berlin, Germany
Mauro Pezz	University of Lugano, Italy
Corneliu Popeea	TU Munich, Germany
Sophie Quinton	TU Braunschweig, Germany
Jan Rutten	CWI, The Netherlands
Geoffrey Smith	Florida International University, USA
Jaco Van De Pol	University of Twente, The Netherlands
Helmut Veith	Vienna University of Technology, Austria
Martin Wirsing	Ludwig Maximilians University of Munich, Germany
Nobuko Yoshida	Imperial College London, UK
Gianluigi Zavattaro	Università di Bologna, Italy

Additional Reviewers

Steering Committee

Jean-Bernard Stefani	(Chair, elected member)
Frank de Boer	(Elected member)
Einar Broch Johnsen	(Elected member)
Heike Wehrheim	(Elected member)
John Hatcliff	(Rotating member, 2010–2013)
Elena Zucca	(Rotating member, 2010–2013)
Roberto Bruni	(Rotating member, 2011–2014)
Juergen Dingel	(Rotating member, 2011–2014)
Holger Giese	(Rotating member, 2012–2015)
Grigore Rosu	(Rotating member, 2012–2015)

Table of Contents

Analyzing Interactions of Asynchronously Communicating Software Components (Invited Paper)

Tevfik Bultan

Department of Computer Science
University of California, Santa Barbara
bultan@cs.ucsb.edu

Abstract. Since software systems are becoming increasingly more concurrent and distributed, modeling and analysis of interactions among their components is a crucial problem. In several application domains, message-based communication is used as the interaction mechanism, and the communication contract among the components of the system is specified semantically as a state machine. In the service-oriented computing domain this type of message-based communication contracts are called "choreography" specifications. A choreography specification identifies allowable ordering of message exchanges in a distributed system. A fundamental question about a choreography specification is determining its realizability, i.e., given a choreography specification, is it possible to build a distributed system that communicates exactly as the choreography specifies? In this short paper we give an overview of this problem, summarize some of the recent results and discuss its application to web service choreographies, Singularity OS channel contracts, and UML collaboration (communication) diagrams.

1 Introduction

Nowadays, many software systems consist of multiple components that execute concurrently on different machines that are physically distributed, and interact with each other through computer networks. Moreover, new trends in computing, such as service-oriented architecture, cloud computing, multi-core hardware, wearable computing, all point to even more concurrency and distribution among the components of software systems in the future. Concurrent and distributed software systems are increasingly used in every aspect of society and in some cases provide safety critical services. Hence, it is very important to develop techniques that guarantee that these software systems behave according to their specifications.

A crucial problem in dependability of concurrent and distributed software systems is the coordination of different components that form the whole system. In order to complete a task, components of a software system have to coordinate their executions by interacting with each other. One fundamental question is, what should be the interaction mechanism given the trend for increased level of concurrency and distribution in computing? One emerging paradigm is message-based communication [2,7,10,8], where components interact with each other by sending and receiving messages. Given these trends, we conclude that analyzing message-based interactions among software components is a timely and significant research problem.

2 Specification of Message-Based Interactions

Specification and analysis of message-based interactions has been an active research area studied in several application domains including coordination in service-oriented computing [7,12], interactions in distributed programs [1] and process isolation at the OS level [8].

Service oriented computing provides technologies that enable multiple organizations to integrate their businesses over the Internet. Typical execution behavior in this type of distributed systems involves a set of autonomous peers interacting with each other through messages. Modeling and analysis of interactions among the peers is a crucial problem in this domain due to following reasons: 1) Organizations may not want to share the internal details of the services they provide to other organizations. In order to achieve decoupling among different peers, it is necessary to specify the interactions among different services without referring to the details of their local implementations. 2) Modeling and analyzing the global behavior of this type of distributed systems is particularly challenging since no single party has access to the internal states of all the participating peers. Desired behaviors have to be specified as constraints on the interactions among different peers since the interactions are the only observable global behavior. Moreover, for this type of distributed systems, it might be worthwhile to specify the interactions among different peers before the services are implemented. Such a top-down design strategy may help different organizations to better coordinate their development efforts.

Choreography languages enable specification of such interactions. A choreography specification corresponds to a global ordering of the message exchange events among the peers participating to a composite service, i.e., a choreography specification identifies the set of allowable message sequences for a composite web service.

3 Choreography Analysis

Specification of interactions in a software system as a choreography leads to several interesting research problems [6]:

– *Realizability:* Given a choreography specification, determining if there exists a set of components that generate precisely the set of message sequences specified by the choreography specification.

- *Synthesis:* Given a choreography specification, synthesizing a set of components that generate precisely the set of message sequences specified by the choreography specification.
- *Conformance:* Determining if a set of given components adhere to a given choreography specification.
- *Synchronizability:* Determining if the set of interactions generated by a given set of components remain the same under asynchronous and synchronous communication.

Some formalizations of these questions lead to unsolvable problems. For example, choreography conformance problem is undecidable when asynchronous communication is used. This is because, systems where peers communicate asynchronously with unbounded FIFO message queues can simulate Turing Machines [5].

It is important to note that the choreography analysis problem is not isolated to the area of service-oriented computing. It is a fundamental problem that appears in any area where message-based communication is used to coordinate interactions of multiple concurrent or distributed components. For example, recently, earlier results on choreography analysis have been applied to analysis of Singularity channel contracts [11]. Singularity is an experimental operating system developed by Microsoft Research in order to improve the dependability of software systems [9]. In the Singularity operating system all inter-process communication is done via messages sent through asynchronous communication channels. Each channel is governed by a channel contract [8]. A channel contract is basically a state machine that specifies the allowable ordering of messages between the client and the server. Hence, channel contracts serve the same purpose that choreography specifications serve in service oriented computing.

4 Recent Results

There has been some recent progress in addressing these research problems. It has been shown that synchronizability checking is decidable [3]. It can be solved by comparing the behavior of a system with synchronous communication to the behavior of the same system with bounded asynchronous communication where the queue sizes are limited to one. This result also leads to effective approaches to choreography conformance checking. Although choreography conformance problem is undecidable in general, synchronizability analysis identifies a class of systems for which choreography conformance can be checked using synchronous communication instead of asynchronous communication. This means that message queues can be removed during the conformance analysis, significantly reducing the state space of the analyzed system.

More recently, it has been shown that choreography realizability problem is decidable for systems communicating with asynchronous messages using unbounded FIFO message queues [4]. This also means that for realizable choreography specifications, synthesis problem can be solved by projecting the given choreography specification to each component that participates to the choreography.

References

1. Armstrong, J.: Getting Erlang to Talk to the Outside World. In: Proc. ACM SIGPLAN Work. on Erlang, pp. 64–72 (2002)
2. Banavar, G., Chandra, T., Strom, R., Sturman, D.: A Case for Message Oriented Middleware. In: Jayanti, P. (ed.) DISC 1999. LNCS, vol. 1693, pp. 1–17. Springer, Heidelberg (1999)
3. Basu, S., Bultan, T.: Choreography Conformance via Synchronizability. In: Proc. 20th Int. World Wide Web Conf. (2011)
4. Basu, S., Bultan, T., Ouederni, M.: Deciding Choreography Realizability. In: Proc. 39th Symp. Principles of Programming Languages (2012)
5. Brand, D., Zafiropulo, P.: On communicating finite-state machines. Journal of the ACM 30(2), 323–342 (1983)
6. Bultan, T., Fu, X., Su, J.: Analyzing conversations: Realizability, synchronizability, and verification. In: Baresi, L., Di Nitto, E. (eds.) Test and Analysis of Web Services, pp. 57–85. Springer (2007)
7. Carbone, M., Honda, K., Yoshida, N., Milner, R., Brown, G., Ross-Talbot, S.: A Theoretical Basis of Communication-Centred Concurrent Programming, W3C Note (October 2006), `http://www.w3.org/2002/ws/chor/edcopies/theory/note.pdf`
8. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G.C., Larus, J.R., Levi, S.: Language support for fast and reliable message-based communication in singularity os. In: Proc. 2006 EuroSys Conf., pp. 177–190 (2006)
9. Hunt, G.C., Larus, J.R.: Singularity: rethinking the software stack. Operating Systems Review 41(2), 37–49 (2007)
10. Menascé, D.A.: Mom vs. rpc: Communication models for distributed applications. IEEE Internet Computing 9(2), 90–93 (2005)
11. Stengel, Z., Bultan, T.: Analyzing Singularity Channel Contracts. In: Proceedings of the 18th International Symposium on Software Testing and Analysis, pp. 13–24 (2009)
12. Web Service Choreography Description Language (WS-CDL) (2005), `http://www.w3.org/TR/ws-cdl-10/`

Formal Analysis of a Distributed Algorithm for Tracking Progress

Martín Abadi[1,2], Frank McSherry[1],
Derek G. Murray[1], and Thomas L. Rodeheffer[1]

[1] Microsoft Research Silicon Valley
[2] University of California, Santa Cruz

Abstract. Tracking the progress of computations can be both important and delicate in distributed systems. In a recent distributed algorithm for this purpose, each processor maintains a delayed view of the pending work, which is represented in terms of points in virtual time. This paper presents a formal specification of that algorithm in the temporal logic TLA, and describes a mechanically verified correctness proof of its main properties.

1 Introduction

In distributed systems, it is often useful and non-trivial to know how far a computation has progressed. In particular, the problem of termination detection is classic and remains important. More generally, distributed systems often need to detect progress—not just complete termination—for the sake of correctness and efficiency. For example, knowing that a broadcast message has reached all participants in a protocol enables the sender to reclaim memory and other resources associated with the message; similarly, establishing that a certain phase of a computation has completed can contribute to resource management, can inform scheduling decisions, and also enables speculative computation steps to commit and to result in visible output. For such tasks, protocols need to aggregate and share the local views of the system components. Those protocols may operate continuously ("on-the-fly"), or be triggered from time to time by the need to reclaim resources or by external events. In either case, they are often interesting, delicate, crucial for correctness, and worthy of careful design and analysis.

We are presently engaged in research on large-scale, data-parallel distributed computation, and on the development of a system for this purpose, called Naiad [9]. This research explores a declarative dataflow model that supports incremental and iterative computations, with a generalization of the notion of virtual time [5]. According to its original definition, virtual time is

> a global, one-dimensional, temporal coordinate system imposed on a distributed computation; it is used to measure progress and to define synchronization. It may or may not have a connection with real time.

Naiad relaxes this notion by allowing the temporal coordinate system to be based on a partial order, rather than a one-dimensional linear order, as already suggested in passing by Jefferson [5, p407]. Thus, inputs and other pieces of data are associated with time points in this partial order. The use of a partial order avoids unnecessary constraints (false dependencies) that can hinder the progress of computations.

As in prior work on virtual time (e.g., [12]), progress detection is essential to Naiad. Accordingly, the research on Naiad to date includes the design and implementation of a new distributed algorithm for progress detection. This algorithm relies on out-of-band communication (rather than on punctuation within message streams; cf. [2], [13]) for continuously tracking the progress of a computation with respect to partially ordered virtual time (cf. [12]).

For the present purposes, the significance of these characteristics is less important than the fact that we are interested in a new distributed algorithm for progress detection, that a system depends on it, and that we therefore wish to understand it as well as possible. The goal of this paper is to develop a rigorous specification and analysis of the algorithm. The paper presents a formal specification of the algorithm and its properties. Furthermore, it describes a complete, mechanically verified correctness proof for the algorithm. This proof is quite long, as we explain below. So we do not describe all its steps in this paper, but we summarize its main definitions and lemmas, and we briefly describe our approach and experience doing the proof.

We use TLA [7], a well established linear-time temporal logic, and its associated tools [4], [6]. The choice of a linear-time formalism is not in contradiction with the study of a notion of virtual time that relies on a partial order, as this work illustrates. TLA enables a concise and general description of the algorithm. Through its tools, TLA also enables the mechanical verification of our proof.

We believe that this work has a number of benefits.

- As intended, our study has increased confidence in the algorithm, its properties, and its suitability for Naiad.
- This study has also resulted in a detailed, rigorous, and abstract explanation of the algorithm. The generality of the formulation of the algorithm enables us to contemplate other applications, in other systems or even in Naiad as this system evolves. For example, we can identify the essential properties of the algorithm related to the partial order, without making unnecessary assumptions (for example, that the partial order is well-founded, that it is a lattice, or that it is a product of linear orders) which might hold only in particular circumstances.
- Finally, since this study constitutes one of the largest and most difficult applications of TLA to distributed algorithms to date, it has led to new experience with TLA and to detailed feedback to the TLA developers (for instance, on libraries and on performance issues in the TLA proof tools). Some of this feedback has already resulted in improvements to the TLA Toolbox [6].

The next section contains a brief review of TLA. Section 3 describes the algorithm and its main properties. Section 4 presents our proof. Section 5 concludes. A companion technical report contains complete details of the formal specification and proof [11].

2 A Brief Review of TLA

TLA (the Temporal Logic of Actions) [7] combines first-order predicate logic, set theory, and linear-time temporal operators. Figure 1 reviews the TLA notations used in the formulas in this paper.

The TLA Toolbox [6] is an integrated development environment for writing and checking TLA specifications. Specification can include theorems along with their proofs. Proofs are written as sequences of proof steps. The Toolbox includes the TLA Proof System [4], which checks proofs: it mechanically verifies each proof step by constructing proof obligations and discharging them via a number of back-end provers. The Toolbox also includes a standard library of definitions about natural numbers, integers, sequences, and finite sets, along with fundamental theorems about induction over natural numbers.

3 The Algorithm

In this section, we describe our algorithm, first informally and then in TLA. We also state its main safety property.

3.1 Informal Description

Our progress-detection algorithm oversees a computation. Each state of this computation includes a multiset of records, and the computation consists of a sequence of operations that act on this multiset: each operation atomically consumes some of the existing records and replaces them with some output records. The operations and the ordering of the computation as a whole may be non-deterministic. In particular, in a dataflow system such as Naiad, the records contain data, they flow through a graph, and the nodes in the graph asynchronously perform the operations.

In any state of a computation, the existing records may correspond to different stages in the logical progress of the computation. As an example, let us consider a computation that consumes records that contain images, and that processes each image by sequentially applying two functions f_1 and f_2. In an intermediate state of the computation, an input record x that has not yet been processed at all may coexist with the result $f_1(y)$ of applying f_1 to another input record y, and with the result $f_2(f_1(z))$ of applying both f_1 and f_2 to a third input record z. One may think of the records as corresponding to different points in virtual time. These points in virtual time indicate the progress of the computation. In this case, three linearly ordered points will suffice. In general, as in this example, Comments in TLA appear shaded like this. Declarations of symbols must appear before the symbols are used, and most of the notation resembles that of ordinary mathematics.

Top-level declarations appear at the beginning of a line and can be used subsequently.

CONSTANT *con*	the constant *con*
VARIABLE *var*	the state variable *var*
$zerop \triangleq$ formula	an operator that takes no arguments
$monop(a) \triangleq$ formula using a	an operator that takes one argument
$a \oplus b \triangleq$ formula using a and b	an infix operator of two arguments

A LET IN formula creates a declaration that can be used within its subformula.

LET $op \triangleq$ op-def-formula IN subformula using op

TLA uses the ordinary symbols of set theory $\in \ \notin \ \cup \cap \{ \ \}$
and of propositional logic $\wedge \vee \neg \Rightarrow = \neq$ with first-order quantifiers $\forall\ \exists$
and the common programming syntax IF THEN ELSE
and has standard libraries for natural numbers and integers $+ - < \leq \geq >$ *Nat Int*.

$\wedge$ subformula-1 $\wedge$ subformula-2 $\vdots$ $\wedge$ subformula-n	A conjunction can be written on a series of lines that all begin with $\wedge$ in the same column. A subformula can extend over multiple lines provided it does not intrude on the column used by its leading $\wedge$. The same syntax works for a disjunction with $\vee$.

TLA has syntax for sets and for functions that map one set to another.

SUBSET A	the set of all subsets of A (powerset)
$\{a \in A : P(a)\}$	the set of all $a \in A$ such that $P(a)$
CHOOSE $a \in A : P(a)$	the arbitrary choice of some $a \in A$ such that $P(a)$
$[A \rightarrow B]$	the set of all functions that map A to B
$[a \in A \mapsto G(a)]$	the function that maps each $a \in A$ to $G(a)$
$[M$ EXCEPT $![d] = F]$	the function the same as M except that d maps to F
$M[a]$	the result of applying M to a
DOMAIN M	the domain of the function M

It is possible to declare a recursive function.

LET $M[a \in A] \triangleq$ def using M and a IN subformula using M

A sequence in TLA is a function that maps $1..n$ to the elements of some set, where $n \in Nat$ is the length of the sequence.

$\langle\rangle$	the empty sequence
$Seq(D)$	the set of all sequences of D
$Len(Q)$	the length of the sequence Q
$Append(Q, d)$	the result of appending d to the sequence Q
$Tail(Q)$	the result of removing the first element from the sequence Q

TLA has several temporal operators, of which we use two.

$\Box P$	P is true now and at all times in the future.
F'	the value of F in the next time step

Fig. 1. Brief review of TLA

we assume a set of points of virtual time, with a partial order, and associate each record with a point in virtual time, but the set of points need not be finite, and the partial order need not be linear.

We do require that, if an operation produces a record at one point in virtual time, then the operation has consumed at least one record at a strictly lower

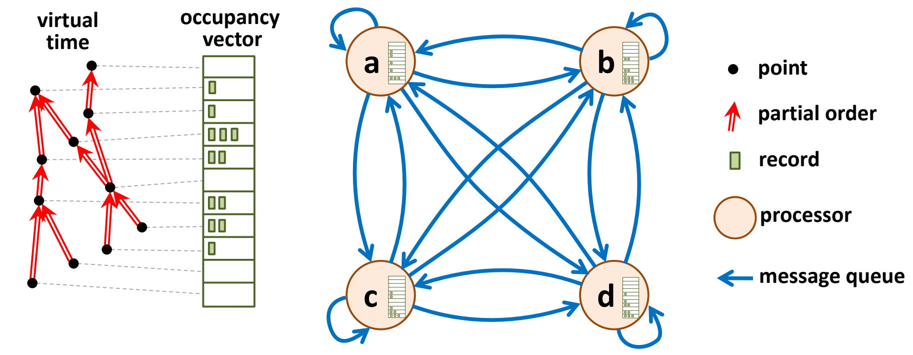


Fig. 2. Overall structure: each processor locally accumulates a delayed view of the occupancy vector

point according to the partial order. Therefore, as a computation proceeds, the population of records will migrate away from lower points. Should a downward-closed set of points become vacant, this set will always thereafter remain vacant, as any operation that might produce a record associated with a point in the set would need to consume such a record as well. (See the safety property *Safe2* in Sect. 4.) This monotonically increasing set of permanently vacant points represents the progress that we wish to track.

We envision that a distributed collection of processors will perform the operations. In such a distributed system, each processor will not be able to observe, directly, the full, exact contents of the set of records in order to measure progress. Processors must instead communicate with one another, as they perform operations, exchanging information about the records that those operations consume and produce. With this information, each processor can maintain a possibly delayed but always safe approximation to the set of permanently vacant points in virtual time.

More concretely, in our algorithm, each processor maintains a local occupancy vector that maps each point to the processor's view of the number of records at that point, depicted in Fig. 2. At the start of a computation, this vector is defined from the initial set of records in the system. A processor tracks the changes in occupancy due to the operations that it performs. When convenient, the processor broadcasts incremental updates to all processors, sending updates about points with net production of records before those about points with net consumption of records. (Section 3.3 describes the exact ordering requirement.) When a processor receives one of these updates, it adjusts its local occupancy vector accordingly. We assume that communication channels between processors are reliable and completely ordered, so that updates are neither dropped nor delivered out of order, and similarly we assume that the processors themselves are reliable; standard communication protocols and fault-tolerance techniques can provide these guarantees.

The intent of this approach is that, once a downward-closed set of points becomes vacant in the local occupancy vector of some processor, that same set

CONSTANT Point	set of points
CONSTANT Proc	set of processors
CONSTANT $\_ \preceq \_$	partial order on Point

$\mathit{CountVec} \triangleq [\mathit{Point} \rightarrow \mathit{Nat}]$	count vectors
$\mathit{DeltaVec} \triangleq [\mathit{Point} \rightarrow \mathit{Int}]$	delta vectors

$Z \triangleq [t \in \mathit{Point} \mapsto 0]$	everywhere zero
$a \oplus b \triangleq [t \in \mathit{Point} \mapsto a[t] + b[t]]$	component-wise addition
$a \ominus b \triangleq [t \in \mathit{Point} \mapsto a[t] - b[t]]$	component-wise subtraction
$s \prec t \triangleq s \preceq t \wedge s \neq t$	strictly lower

$$\begin{aligned}
&\mathit{IsVacantUpto}(a,\ t) &&\triangleq \forall\, s \in \mathit{Point} : s \preceq t \Rightarrow a[s] = 0\\
&\mathit{IsNonposUpto}(a,\ t) &&\triangleq \forall\, s \in \mathit{Point} : s \preceq t \Rightarrow a[s] \leq 0\\
&\mathit{IsSupported}(a,\ t) &&\triangleq \exists\, s \in \mathit{Point} : s \prec t \wedge a[s] < 0 \wedge \mathit{IsNonposUpto}(a,\ s)\\
&\mathit{IsUpright}(a) &&\triangleq \forall\, t \in \mathit{Point} : a[t] > 0 \Rightarrow \mathit{IsSupported}(a,\ t)
\end{aligned}$$

VARIABLE nrec	$\in \mathit{CountVec}$
VARIABLE temp	$\in [\mathit{Proc} \rightarrow \mathit{DeltaVec}]$
VARIABLE msg	$\in [\mathit{Proc} \rightarrow [\mathit{Proc} \rightarrow \mathit{Seq}(\mathit{DeltaVec})]]$
VARIABLE glob	$\in [\mathit{Proc} \rightarrow \mathit{DeltaVec}]$

Fig. 3. Basic definitions

of points is in fact vacant thereafter in the global set of records. (We state this safety property formally in Sect. 3.4.) Although the local occupancy vector can be a delayed view of the true occupancy vector, it is a safe approximation, so it allows each processor to report correct results from completed parts of the computation to external observers; it is also a useful input to each processor's memory management and scheduling decisions.

3.2 Basic Definitions

The formal specification of the progress-detection algorithm starts with the basic definitions shown in Fig. 3.

Those definitions introduce three constants: *Point* is the set of points, *Proc* is the set of processors, and $\preceq$ is a partial order on *Point*. There is no requirement that either *Point* or *Proc* be finite.

According to the definitions, a count vector maps each point to a natural number, which represents a count of the number of records at that point. Similarly, a delta vector represents a change in record counts per point. We use Z to designate the delta vector that is everywhere zero and $\oplus$ and $\ominus$ to indicate component-wise addition and subtraction.

We say that a point t in a delta vector a is negative iff $a[t] < 0$, and positive iff $a[t] > 0$. Describing the relative locations of positive and negative points in delta vectors is essential to our proof, so we define several predicates for this purpose. A delta vector a is vacant up t iff $a[s] = 0$ for all $s \preceq t$; it is non-positive up t iff $a[s] \leq 0$ for all $s \preceq t$. A delta vector a is supported at point t iff there exists a negative point $s \prec t$ such that a is non-positive up to s. We then say that s supports t. A delta vector is upright iff all of its positive points are supported.

This definition of upright delta vectors arises because we use delta vectors to describe the changes in record counts that operations cause. As indicated in Sect. 3.1, we require that for any point t at which an operation causes a net production of records there must be a lower point s at which the operation causes a net consumption of records; this property explains why, in an upright delta vector, for each positive point t there must exist a negative point $s \preceq t$. For s to support t, we further require that all points $u \preceq s$ be non-positive; this property prevents cases of infinite descent. It yields, in particular, that the sum of two upright delta vectors is upright. (In cases where $\preceq$ is well-founded, infinite descent is impossible, so the further requirement becomes superfluous.)

Finally, we specify the state of the algorithm using four state variables: *nrec*, *temp*, *msg*, and *glob*.

- *nrec* is the occupancy vector, which represents the number of records that currently exist at each point.
- *temp*$[p]$ is the local (temporary) change in the occupancy vector due to the performance of operations at processor p. Note that the change at a given point can be negative (net records consumed), positive (net records produced), or zero. We call it temporary because eventually the processor takes the information from *temp*$[p]$ and broadcasts it as an incremental update.
- *msg*$[p][q]$ is the queue of updates from processor p to processor q. Each update is a delta vector that is zero everywhere except at those points that contain information about net changes. Implementations may of course limit the number of non-zero points and represent updates in a compact form.
- *glob*$[q]$ is the delayed view at processor q of the occupancy vector. It is a delta vector, rather than a count vector, because *glob*$[q][t]$ can be negative for some point t. Such negative values can appear, for example, when one processor p_1 produces a record at point t, a second processor p_2 consumes it and, because of different queuing delays, processor q receives the update from p_2 before that from p_1.

3.3 The Algorithm

Building on the definitions of Fig. 3, Fig. 4 gives the specification of the progress-detection algorithm. It defines an initial condition *Init*, a next-state relation *Next*, and then a complete specification *Spec* which states that *Init* must hold and then forever each step must satisfy the *Next* relation.

Init states that *nrec* can be any mapping from *Point* to *Nat*; this mapping represents an arbitrary initial population of records. Initially, there are no unsent changes, no unreceived updates, and each processor knows the initial population.

Each step from a current state to a next state is an action specified as a relation between the values of the state variables in the current state (unprimed) and in the next state (primed). The algorithm has three actions: *NextPerformOperation*, *NextSendUpdate*, and *NextReceiveUpdate*.

- In the *NextPerformOperation* action, processor p performs an operation that consumes and produces some number of records at each point. The records

```
$Init \triangleq$
  $\land\ nrec \in CountVec$                                  any initial population of records
  $\land\ temp = [p \in Proc \mapsto Z]$                        no unsent changes
  $\land\ msg = [p \in Proc \mapsto [q \in Proc \mapsto \langle\rangle]]$   no unreceived updates
  $\land\ glob = [q \in Proc \mapsto nrec]$                     each processor knows the initial nrec

$NextPerformOperation \triangleq \exists\, p \in Proc,\ c \in CountVec,\ r \in CountVec :$
   LET $delta \triangleq r \ominus c$ IN                  the net change in record population
    $\land\ \forall\, t \in Point : c[t] \leq nrec[t]$     only consume what exists
    $\land\ IsUpright(delta)$                              net change must be upright
    $\land\ nrec' = nrec \oplus delta$
    $\land\ temp' = [temp$ EXCEPT $![p] = temp[p] \oplus delta]$
    $\land$ UNCHANGED $msg$
    $\land$ UNCHANGED $glob$

$NextSendUpdate \triangleq \exists\, p \in Proc,\ tt \in$ SUBSET $Point :$
   LET $gamma \triangleq [t \in Point \mapsto$ IF $t \in tt$ THEN $temp[p][t]$ ELSE $0]$ IN
    $\land\ gamma \neq Z$                                 update must say something
    $\land\ IsUpright(temp[p] \ominus gamma)$             what is left must be upright
    $\land$ UNCHANGED $nrec$
    $\land\ temp' = [temp$ EXCEPT $![p] = temp[p] \ominus gamma]$
    $\land\ msg' = [msg$ EXCEPT $![p] = [q \in Proc \mapsto Append(msg[p][q],\ gamma)]]$
    $\land$ UNCHANGED $glob$

$NextReceiveUpdate \triangleq \exists\, p \in Proc,\ q \in Proc :$
   LET $kappa \triangleq msg[p][q][1]$ IN               oldest unreceived update from p to q
    $\land\ msg[p][q] \neq \langle\rangle$                message queue must be non-empty
    $\land$ UNCHANGED $nrec$
    $\land$ UNCHANGED $temp$
    $\land\ msg' = [msg$ EXCEPT $![p][q] = Tail(msg[p][q])]$
    $\land\ glob' = [glob$ EXCEPT $![q] = glob[q] \oplus kappa]$

$Next \triangleq NextPerformOperation \lor NextSendUpdate \lor NextReceiveUpdate$

$Spec \triangleq Init \land \Box Next$
```

Fig. 4. Formal specification of the progress-detection algorithm

to be consumed must exist and the net change in records *delta* must be an upright delta vector. The action adds *delta* to *nrec* and to $temp[p]$.

- In the *NextSendUpdate* action, processor p selects a set of points tt and broadcasts an update about its changes at those points. The update is represented by *gamma*. The processor must choose tt in such a way that $temp[p] \ominus gamma$ is upright. This requirement holds, in particular, when tt consists of positive points in $temp[p]$ if any exist, because $temp[p]$ is always upright, as we show in Sect. 4. The action subtracts *gamma* from $temp[p]$ and appends *gamma* to $msg[p][q]$ for all q.
- In the *NextReceiveUpdate* action, processor q selects a processor p and receives the oldest update *kappa* on the message queue from p to q. For this action to take place, the current message queue $msg[p][q]$ must be non-empty. The action adds *kappa* to $glob[q]$ and removes it from $msg[p][q]$.

For any point t and processor q, if $glob[q]$ is vacant up to t, then, at this and all future times, $nrec$ is vacant up to t.

$Safe \triangleq \forall\, t \in Point,\ q \in Proc :\ (\ IsVacantUpto(glob[q], t) \Rightarrow \Box IsVacantUpto(nrec, t)\ )$

An execution that obeys *Spec* is always *Safe*.

THEOREM $Spec \Rightarrow \Box Safe$

Fig. 5. Main safety property of the progress-detection algorithm

The next-state relation *Next* is simply the disjunction of the relations for these three actions.

An implementation may refine this specification in many ways. In particular, while each of the three actions in the specification is atomic, an implementation may perform smaller steps. An implementation may also exhibit less nondeterminism, for example by restricting the choice of *tt* in *NextSendUpdate*. As usual, our results automatically carry over to all correct implementations of the specification.

3.4 The Main Safety Property

The main goal of the progress-detection algorithm is to enable each processor q to deduce global information about the present and future of *nrec* from current, local information about $glob[q]$, as explained in Sect. 3.1. Specifically, if $glob[q]$ is vacant up to t then *nrec* should also be vacant up to t at this and all future times. The main correctness property of the algorithm is that this implication always holds. Figure 5 expresses the implication as a TLA formula *Safe*. Our main theorem is that $\Box Safe$ holds in every execution that obeys *Spec*.

In addition to this safety property, the algorithm satisfies a liveness property: if progress happens then it will eventually be known to all processors, under suitable fairness assumptions on the transmission of updates and other actions. We regard this liveness property as important but less crucial than the safety property, and its treatment would require additional notations, definitions, and arguments, so we do not consider it further in this paper.

4 Formal Verification (Summary)

In this section, we present our formal proof of the correctness of the progress-detection algorithm, that is, of the theorem $Spec \Rightarrow \Box Safe$ stated in Fig. 5. As indicated in the Introduction, the proof is quite long, so we summarize it, describing its argument and stating key auxiliary safety properties. We also comment on various characteristics of the proof and on our experience.

4.1 Proof Summary

As usual, invariants are predicates on states that always hold in every execution that obeys the specification, and we talk about "proving an invariant" when

For any point t, if *nrec* is vacant up to t, then it will be so at all future times.
$\mathit{Safe2} \triangleq \forall\, t \in \mathit{Point} : (\ \mathit{IsVacantUpto}(\mathit{nrec}, t) \Rightarrow \Box \mathit{IsVacantUpto}(\mathit{nrec}, t)\ )$

For any point t and processor q, if $\mathit{glob}[q]$ is vacant up to t, then so is *nrec*.
$\mathit{Inv1} \triangleq \forall\, t \in \mathit{Point},\ q \in \mathit{Proc} : (\ \mathit{IsVacantUpto}(\mathit{glob}[q], t) \Rightarrow \mathit{IsVacantUpto}(\mathit{nrec}, t)\ )$

Fig. 6. Auxiliary safety property and invariant

what we mean is proving that the state predicate in question is in fact an invariant. To prove an invariant I, it suffices to show that the invariant holds in the initial state ($\mathit{Init} \Rightarrow I$) and that every step that satisfies the next-state relation maintains the invariant ($I \wedge \mathit{Next} \Rightarrow I'$). It follows by induction that I is true in every reachable state. Proving a general safety property is more complicated, but similar deduction rules apply.

The safety property *Safe* follows from an auxiliary safety property *Safe2* and an invariant *Inv1* defined in Fig. 6. *Safe2* says that, whenever *nrec* is vacant up to some point t, it will stay that way. This safety property is a simple consequence of the two requirements $\forall\, t \in \mathit{Point} : c[t] \leq \mathit{nrec}[t]$ and $\mathit{IsUpright}(\mathit{delta})$ in *NextPerformOperation*, which is the only action that changes *nrec*. *Inv1* says that, for any q, if $\mathit{glob}[q]$ is vacant up to some point t, then so is *nrec*. We devote the rest of this summary to explaining the proof of *Inv1*, which is much harder than that of *Safe2*.

In order to prove *Inv1*, we consider the relation between *nrec*, $\mathit{glob}[q]$, and all of the information about changes to *nrec* that has not yet been incorporated into $\mathit{glob}[q]$. For this purpose, we define $\mathit{Info}(k, p, q)$ as the suffix of information from processor p heading toward processor q that skips the k oldest unreceived updates, and $\mathit{AllInfo}(q)$ as the sum of all information heading toward processor q. Figure 7 introduces the necessary definitions and a lemma, and Fig. 8 asserts several additional auxiliary invariants. Next we discuss the lemma, the auxiliary invariants, and the derivation of *Inv1*, summarizing informally the reasoning that our proof makes formally:

– The lemma states that the sum of two upright delta vectors a and b is upright. This lemma is proved by a case analysis on the location of positive and negative points in the sum. A positive point t in $a \oplus b$ can occur only where at least one of a or b is positive. Without loss of generality, let $a[t] > 0$. Then, since a is upright, there must be a point s that supports t in a. It follows that $a[s] < 0$ and a is non-positive up to s. If b is non-positive up to s then s supports t in $a \oplus b$. Otherwise, there is some $u \preceq s$ such that $b[u] > 0$. Since b is upright, there must be a point v that supports u in b. We can conclude that $(a \oplus b)[v] < 0$ and $a \oplus b$ is non-positive up to v, so v supports t in $a \oplus b$. In either case, there exists a point that supports t in $a \oplus b$. Therefore, $a \oplus b$ is upright.
– *Inv2* states that the sum of $\mathit{glob}[q]$ plus all information heading toward q is *nrec*. This predicate is trivially true in the initial state. *NextPerformOperation* transfers *delta* from *nrec* to $\mathit{temp}[p]$; *NextSendUpdate* transfers

Sum of the sequence Q of delta vectors, skipping the first k. This operator is defined for all $k \in Nat$; the result is Z when $k \geq Len(Q)$. A recursive function computes the sum.

```
$SumSeq(k, Q) \triangleq$
    LET $Elem(i) \triangleq$ IF $k < i \land i \leq Len(Q)$ THEN $Q[i]$ ELSE $Z$ IN
    LET $Sumv[i \in Nat] \triangleq$ IF $i = 0$ THEN $Z$ ELSE $Sumv[i-1] \oplus Elem(i)$ IN
    $Sumv[Len(Q)]$
```

Given a finite set choose a sequence in which each element appears exactly once.

```
$ExactSeqFor(D) \triangleq$ CHOOSE $Q \in Seq(D)$ :
    $\land \forall d \in D : \exists i \in$ DOMAIN $Q : Q[i] = d$           each element appears
    $\land \forall i, j \in$ DOMAIN $Q : Q[i] = Q[j] \Rightarrow i = j$     there are no duplicates
```

Sum of the delta vectors in the range of F, assuming only finitely many are non-zero.

```
$SumFun(F) \triangleq$ LET $I \triangleq ExactSeqFor(\{d \in$ DOMAIN $F : F[d] \neq Z\})$ IN
    $SumSeq(0, [i \in$ DOMAIN $I \mapsto F[I[i]]])$
```

The suffix of information from p heading toward q that skips the k oldest unreceived updates. The sum includes $temp[p]$, which is information that p knows but has not yet sent.

$Info(k, p, q) \triangleq SumSeq(k, msg[p][q]) \oplus temp[p]$

All information heading toward q.

$AllInfo(q) \triangleq SumFun([p \in Proc \mapsto Info(0, p, q)])$

The sum of upright delta vectors is upright.

LEMMA $\forall a, b \in DeltaVec : IsUpright(a) \land IsUpright(b) \Rightarrow IsUpright(a \oplus b)$

Fig. 7. Additional definitions and lemma

For any processor q, the sum of $glob[q]$ plus all information heading toward q is $nrec$.

$Inv2 \triangleq \forall q \in Proc : nrec = glob[q] \oplus AllInfo(q)$

For any processor p, $temp[p]$ is upright.

$Inv3 \triangleq \forall p \in Proc : IsUpright(temp[p])$

For any processors p and q, any suffix of information from p heading toward q is upright.

$Inv4 \triangleq \forall k \in Nat, p \in Proc, q \in Proc : IsUpright(Info(k, p, q))$

For any processor q, non-zero information is coming from only a finite set of processors.

$Inv5 \triangleq \forall q \in Proc : IsFiniteSet(\{p \in Proc : Info(0, p, q) \neq Z\})$

For any processor q, all information heading toward q is upright.

$Inv6 \triangleq \forall q \in Proc : IsUpright(AllInfo(q))$

For any point t, $nrec[t] \geq 0$.

$Inv7 \triangleq \forall t \in Point : nrec[t] \geq 0$

Fig. 8. Additional auxiliary invariants

For any point t, if $glob[q]$ is vacant up to t, then it will be so at all future times.

$Safe3 \triangleq \forall q \in Proc, t \in Point : (IsVacantUpto(glob[q], t) \Rightarrow \Box IsVacantUpto(glob[q], t))$

Fig. 9. An extra safety property proved in the full proof

gamma from $temp[p]$ to $msg[p][q]$; and *NextReceiveUpdate* transfers *kappa* from $msg[p][q]$ to $glob[q]$. Each of these actions maintains *Inv2*.

- *Inv3* states that $temp[p]$ is upright. This predicate is true in the initial state because $IsUpright(Z)$ holds. *NextPerformOperation* adds an upright delta vector to $temp[p]$; *NextSendUpdate* requires $IsUpright(temp[p]')$; and *NextReceiveUpdate* has no effect on $temp[p]$.
- *Inv4* states that $Info(k, p, q)$ is upright. This predicate is trivially true in the initial state. *NextPerformOperation* increases $Info(k, p, q)$ by an upright delta vector. *NextReceiveUpdate* shifts everything one position forward in message queue $msg[p][q]$, resulting in $Info(k, p, q)' = Info(k + 1, p, q)$. Both of these actions maintain *Inv4*. The only complicated case is that of *NextSendUpdate*, which transfers *gamma* from $temp[p]$ to the end of $msg[p][q]$. For $k \leq Len(msg[p][q])$, this action makes no change in $Info(k, p, q)$. For $k > Len(msg[p][q])$, it results in $Info(k, p, q)' = temp[p]'$, so it maintains *Inv4* by *Inv3*.
- *Inv5* states that the set of processors with non-zero information heading toward processor q is finite. This predicate is trivially true in the initial state and is clearly maintained by each action.
- *Inv6* states that $AllInfo(q)$ is upright. This invariant follows easily from *Inv4* and *Inv5*.
- *Inv7* states that $nrec[t] \geq 0$ for all t. This predicate is trivially true in the initial state and is maintained by the requirement $c[t] \leq nrec[t]$ in *NextPerformOperation*, which is the only action that changes *nrec*.

Finally, we derive *Inv1* from *Inv2*, *Inv6*, and *Inv7* via a proof by contradiction, as follows. Given a point t such that $glob[q]$ is vacant up to t, suppose that there was a point $s \preceq t$ such that $nrec[s] \neq 0$. By *Inv7*, we would have $nrec[s] > 0$. By *Inv2*, we would have $AllInfo(q)[s] > 0$. By *Inv6*, $AllInfo(q)$ is upright, so there would be a point $u \preceq s$ such that $AllInfo(q)[u] < 0$. But then, by *Inv2*, we would have $nrec[u] < 0$, which is impossible by *Inv7*. Hence there can be no such point s.

4.2 Discussion

Our mechanically verified formal proof contains many more steps and much more detail than the summary presented above. We also went on to prove an extra safety property *Safe3* (shown in Fig. 9), which states that whenever $glob[q]$ is vacant up to t it stays that way. The proof of *Safe3* requires its own additional set of supporting definitions and auxiliary properties.

The entire proof is quite long, requiring 208 pages. There are several reasons for this length.

- We did not try to shorten the proof, e.g., by factoring out lemmas afterwards.
- We often had to decompose proof steps, manually, into several smaller steps that could be verified by one of the proof system's back-end provers.
- The proof contains material that logically should belong in general libraries.

Table 1. Statistics of the formal proof

				run time	**obligations proved by**		
	modules	**theorems**	**lines**	**(sec)**	**isabelle**	**smt3**	**zenon**
library	8	75	4848	1945	38	196	1391
specific	19	71	5895	2450	20	86	1258
Total	**27**	**146**	**10743**	**4395**	**58**	**282**	**2649**

The TLA proof system consists of a proof manager, which parses the TLA files, constructs a set of proof obligations for each proof step, and then calls on various back-end provers to verify each proof obligation. By default, the proof manager first invokes Zenon [1], a tableau prover for classical first-order logic with equality. Zenon is generally quick to solve simple problems, but tends to fail on anything complicated. If Zenon fails, the proof manager then invokes Isabelle [10] using a specialized TLA object logic that includes propositional and first-order logic, elementary set theory, functions, and the construction of natural numbers. Pragmas can be used to direct the proof manager to appeal to other back-end provers. An entire category of provers based on Satisfiability Modulo Theory (SMT) is especially good at some hard problems that involve arithmetic, uninterpreted functions, and quantifiers. The back-end prover smt3 uses one such SMT prover [8].

In an early phase of our research, Leslie Lamport experimented with the use of SMT provers for establishing properties of delta vectors. He obtained some elegant, succinct proofs, which encouraged our effort. As our research progressed, we generalized the specification and weakened the hypotheses; our arguments became longer. In all, we spent about four man-months writing, revising, and debugging the entire formal proof.

To enable the proof manager to deal with the total number of proof obligations, we divided the proof into separate modules. Table 1 gives some statistics. We classify the modules into those that logically could be considered as general library modules and those whose applicability is limited to this specific proof. The entire proof can be checked in less than two hours on an 2.67 GHz Intel® Core$^{\mathrm{TM}}$ i7 with 4 GB of RAM. Zenon checks almost 90% of the proof obligations; this fact confirms the effectiveness of Zenon. Just over half of the proof obligations appear in modules that could be considered as library modules.

It was particularly challenging to construct and to study the TLA definitions of what it means to sum up the suffix of a sequence of delta vectors and of what it means to sum up the delta vectors in the range of a function. For this purpose, we had to introduce many theorems with formal proofs about these sums and how they are affected by changes in the underlying sequence or function. For example, given $k \in Nat$, $Q \in Seq(DeltaVec)$, $k \leq Len(Q)$, and $a \in DeltaVec$, we have

$$SumSeq(k, Append(Q, a)) = SumSeq(k, Q) \oplus a$$

Establishing such properties was quite tedious. To a human, on the other hand, these properties are obvious consequences of the fact that $\oplus$ is a commutative and associative operator with an identity and is closed over *DeltaVec*. In other words, $(\mathit{DeltaVec}, \oplus)$ is a commutative monoid.

In fact, the definitions of *SumSeq* and *SumFun* and related theorems could be written in a general form as a library with application to any commutative monoid. We attempted to construct such a library, but soon discovered a performance bug in the proof manager. The problem was that our desired library of theorems created a collection of TLA modules with an exponential explosion of paths through nested module dependencies. The proof manager ended up spending an enormous amount of time exploring these paths to match proof obligations against known facts.

We presented an example of such behavior to the implementors of the proof manager. Eventually, Damien Doligez solved the performance problem, improving the TLA proof system so that it could handle the library structure we had desired. However, by then, we had completed our proof using specialized *SumSeq* and *SumFun* theorems in a linear chain of modules.

Another, more superficial limitation in the current TLA proof system is that it does not directly capture temporal reasoning. Specifically, for an invariant I, the proof system can check proofs of $\mathit{Init} \Rightarrow I$ and $I \wedge \mathit{Next} \Rightarrow I'$, but the trivial, routine step from these formulas to $\mathit{Spec} \Rightarrow \Box I$ remains manual.

Despite such difficulties, constructing a formal proof with the support of the TLA tools enabled us to examine closely the details of the progress-detection algorithm and to refine them. In particular, in our first attempt at the proof we relied on a well-founded partial order. As our research advanced, we realized that we could remove the requirement of well-foundedness from the partial order and instead introduce the more liberal concept of uprightness for delta vectors. Similarly, we originally had restrictive requirements on updates and on their ordering. In the course of the proof, we realized that the essential requirement is that $\mathit{temp}[p]$ must always be upright. Such refinements enable a broad range of useful implementations.

5 Conclusion

Formal specifications and correctness proofs are often beneficial in the study of distributed algorithms; the work presented in this paper is not an exception in this respect. Among the many other examples in this area, some pertain to tasks related to progress detection. For instance, Chandy and Misra developed a concise, manual correctness argument for an algorithm for termination detection, in Unity [3]. However, we are not aware of any work directly analogous to ours. In particular, the early research on the Time Warp mechanism includes a correctness argument for an algorithm for estimating a global virtual time (with a linear order) [12]; that argument was relatively brief, not fully formal, and manual.

Therefore, beyond its intrinsic results, we regard our analysis as an informative datapoint on the use of formal specifications and proof tools. Although we might wish that our work had been easier, we did complete it with reasonable effort. Ongoing improvements in tools should simplify similar work in the future.

Acknowledgments. We are grateful to Paul Barham, Damien Doligez, Rebecca Isaacs, Michael Isard, Leslie Lamport, and Stephan Merz for discussions about Naiad, the progress tracking protocol, and the TLA proof system.

References

1. Bonichon, R., Delahaye, D., Doligez, D.: Zenon: An extensible automated theorem prover producing checkable proofs. In: Dershowitz, N., Voronkov, A. (eds.) LPAR 2007. LNCS (LNAI), vol. 4790, pp. 151–165. Springer, Heidelberg (2007)
2. Chandramouli, B., Goldstein, J., Maier, D.: On-the-fly progress detection in iterative stream queries. Proc. VLDB Endow. 2(1), 241–252 (2009)
3. Chandy, K.M., Misra, J.: Proofs of distributed algorithms: An exercise. In: Hoare, C.A.R. (ed.) Developments in Concurrency and Communication, pp. 305–332. Addison-Wesley, Boston (1990)
4. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: Verifying safety properties with the TLA+ proof system. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 142–148. Springer, Heidelberg (2010)
5. Jefferson, D.R.: Virtual time. ACM Trans. Program. Lang. Syst. 7(3), 404–425 (1985)
6. Lamport, L.: The TLA Toolbox, `http://research.microsoft.com/en-us/um/people/lamport/tla/toolbox.html`
7. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
8. Merz, S., Vanzetto, H.: Automatic verification of TLA+ proof obligations with SMT solvers. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 289–303. Springer, Heidelberg (2012)
9. Naiad: Web page, `http://research.microsoft.com/en-us/projects/naiad/`
10. Paulson, L.C.: Isabelle: A Generic Theorem Prover. LNCS, vol. 828. Springer, Heidelberg (1994)
11. Rodeheffer, T.L.: The Naiad clock protocol: Specification, model checking, and correctness proof. Tech. Rep. MSR-TR-2013-20, Microsoft Research, Redmond (February 2013), `http://research.microsoft.com/apps/pubs/?id=183826`
12. Samadi, B.: Distributed Simulation, Algorithms and Performancs Analysis. Ph.D. thesis, University of California, Los Angeles (1985), Tech. Rep. CSD-850006, `http://ftp.cs.ucla.edu/tech-report/198_-reports/850006.pdf`
13. Tucker, P.A., Maier, D., Sheard, T., Fegaras, L.: Exploiting punctuation semantics in continuous data streams. IEEE Trans. Knowl. Data Eng. 15(3), 555–568 (2003)

A Case Study in Formal Verification Using Multiple Explicit Heaps

Wojciech Mostowski

Formal Methods and Tools, University of Twente, The Netherlands
w.mostowski@utwente.nl

Abstract. In the context of the KeY program verifier and the associated Dynamic Logic for Java we discuss the first instance of applying a generalised approach to the treatment of memory heaps in verification. Namely, we allow verified programs to simultaneously modify several different, but possibly location sharing, heaps. In this paper we detail this approach using the Java Card atomic transactions mechanism, the modelling of which requires two heaps to be considered simultaneously – the basic and the transaction backup heap. Other scenarios where multiple heaps emerge are verification of real-time Java programs, verification of distributed systems, modelling of multi-core systems, or modelling of permissions in concurrent reasoning that we currently investigate for KeY. On the implementation side, we modified the KeY verifier to provide a general framework for dealing with multiple heaps, and we used that framework to implement the formalisation of Java Card atomic transactions. Commonly, a formal specification language, such as JML, hides the notion of the heap from the user. In our approach the heap becomes a first class parameter (yet transparent in the default verification scenarios) also on the level of specifications.

1 Introduction

In the formal verification of object-oriented programs the verification tools and associated logics are constantly improved and developed to handle new verification challenges and to deal with larger and more complex programs. Some of these challenges are efficient reasoning about linked data structures [7,14] or concurrent programs [11,15,1]. Central in all these efforts is the notion of the object heap that is used in respective logics to represent the memory that programs operate on and to handle possible object aliasing. In particular, Separation Logic [22], that some of the verification systems utilise, is strictly built around the notion of the heap, rather than the program that operates on it.

In a similar spirit, the KeY system[1] [2], an interactive verifier for Java programs, was recently redesigned and reimplemented to introduce explicit heap representation to the Java Dynamic Logic [23]. Previously, the heap was represented in the KeY logic implicitly through special semantics of object field

[1] http://www.key-project.org

updates with complex built-in rewrite rules [2, Chap. 3]. In the new version the heap is explicit, directly accessible through a dedicated program variable `heap`. In particular, this change in heap treatment enables reasoning with dynamic frames [12] in KeY. To accommodate dynamic frames style of specifications KeY uses an extended version of the Java Modelling Language (JML) [4], named JML*. In the proof obligation generation process the framing conditions expressed in JML* are translated to Java Dynamic Logic by constructing appropriate formulae over the heap program variable.

In the default scenario a Java program operates on just one main heap and so does the reasoning system when such programs are verified. Any specification elements, like the JML assignable clauses that express framing conditions, implicitly refer to this one heap. For example, "**assignable** `o.f, o.g;`" states that fields `f` and `g` of object `o` might be modified on the heap. There are, however, scenarios with computation models that refer to more than one heap. The first and the simplest example is in distributed programs, where some method may modify a set of locations locally as well as remotely through a remote call. For example, such a method could have this framing specification:

```
assignable_local o.f, o.g;
assignable_remote o.g, o.h;
```

stating that an object `o` that is stored both locally and remotely (by holding a copy) is modified partly here and partly there. Note that this notion of multiple heaps is different from Separation Logic, which also talks about several heaps. In Separation Logic a heap can be split into two or more separate heaps with *disjoint* locations. Here we consider heaps that may (but do not have to) share common locations, i.e., one heap may be a (partial) copy of another. In particular, one location can be changed simultaneously on two or more heaps. Generally, different heaps represent different memory sites, either real physical ones or ones introduced to the reasoning system for modelling certain properties.

Our particular use case for considering more than one heap emerged during the rework of the support for Java Card atomic transactions [25, Chap. 7] in KeY to follow the new explicit heap model. The Java Card technology [6] provides a platform to program smart cards with a considerably stripped off version of Java with no concurrency, floating point numbers, or dynamic class loading. The lack of these features along with the security sensitive application areas of smart cards in the financial sector (bank cards), telecommunications (SIM cards), or identity (e.g., electronic passports), used to make Java Card an ideal verification target for many verification tools [24,16,5]. However, one complicating factor in Java Card that was initially *overseen* by researchers is the said atomic transaction mechanism. The KeY system was the first verification tool to fully formalise the details of Java Card transactions and provide a working implementation [3,18].

In short, the transaction mechanism provides a way to group assignments into atomic blocks to preserve the consistency of heap data, which by default in Java Card physically resides in the permanent EEPROM memory. Furthermore, it provides mechanisms to make exceptions to the transaction data roll-back rules as well as to change the default target memory for heap data to reside in the volatile RAM instead. The core of our formalisation of this in Java Dynamic Logic is the simultaneous use of two heap variables in the computation model. The first represents the regular heap. The second one is used to store the backup copy of the heap for the case when the transaction needs to be aborted and the contents of the heap restored. The assignment rules in the logic operate on both heaps at the same time, raising the need to specify framing conditions for these two heaps in JML*. In effect, the heap becomes a specification parameter, a simple example for heap-parametric frame specification would be:

```
    assignable[heap] o.f, o.g;
    assignable[backupHeap] o.f;
```

In the remainder of the paper we explain these ideas using the formalisation of the Java Card transaction mechanism in KeY as a case study. This consists of the core formalisation discussed in Sect. 2 and the extensions necessary for modular reasoning discussed in Sect. 3. The use of two heaps is not the only possible way to formalise Java Card transactions. However, the solution with two heaps provides a very clean formalisation with little implementation overhead (discussed in Sect. 4), especially compared to our previous work [3], and gives a uniform framework to apply our ideas in other verification domains. We discuss this in Sect. 5. Finally, we conclude the paper in Sect. 6. The rest of this section is the relevant background information about Java Card [6,25] and the Java Dynamic Logic [2,23].

Java Card. The Java Card technology provides means to program smart cards with Java. The technology consists of a language specification, which defines the subset of permissible Java in the context of smart cards, a Virtual Machine specification, which defines the semantics of the Java byte-code when run on a smart card, and finally the API, which provides access to the specific routines usually found on smart cards. The complicating feature of Java Card is that programs directly operate on two memories built into the card chip. Any data allocated in the EEPROM memory is *persistent* and kept between card sessions, the data that resides in RAM is *transient* and always lost on card power-down. The memory allocation rules are: (i) all local variables are transient, (ii) all newly created objects and arrays are by default persistent, and (iii) when allocated with a dedicated API call any array (but not an object) can be made transient. Note the important difference between a reference to an object and the actual object contents. While the object fields are stored in the persistent memory, the reference to that object can be kept in a local variable and be transient itself.[2] Any Java variable, once allocated in its target memory, is transparent to the programmer from the syntax point of view, and it is only the underlying Java Card VM that takes appropriate actions according to the memory type associated with the object.

Objects allocated in EEPROM provide the only permanent storage to an application. To maintain consistency of data in EEPROM, Java Card offers the

[2] A garbage collector is not obligatory in Java Card either. Careless handling of references actually leads to memory leaks, something that is often addressed in Java Card programming guidelines [21].

atomic transaction mechanism accessed through the API. The following is a brief, but complete summary of the transaction rules. Updating a single object field or array element is always atomic. Updates can be grouped into transaction blocks, a static API call to `beginTransaction` opens such a block, which is ended by a `commitTransaction` call, an explicit `abortTransaction` call, or an implicit abort caused by an unexpected program termination (e.g., card power loss). A commit guarantees that all the updates in the block are executed in one atomic step. An abort reverts the contents of the *persistent memory* to the state before the transaction was entered. Note that an explicit abort does not terminate the whole application, only cancels out persistent updates from within the transaction and the program continues its execution. Finally, the API provides so-called *non-atomic* methods to bypass the transaction mechanism. A non-atomic update of a *persistent* array element is never cancelled out by an abort, provided the same array was not manipulated with regular assignments earlier in the same transaction. We provide illustrative examples for these rules later in Sect. 2.

Java Dynamic Logic with Explicit Heap. The Java Dynamic Logic (JDL) [2,23] of the KeY system is an instance of Dynamic Logic [8] tailored to Java. Modalities $\langle p\rangle\phi$ and $[p]\phi$ represent the notion of total and partial correctness, respectively, of program p w.r.t. property ϕ. Java programs are deterministic, hence total correctness requires p's termination, including the absence of top-level exceptions, for partial correctness termination is not required. Formula ϕ is built from logic terms using the usual connectives. Terms contain references to *logic* variables – rigid symbols whose valuation is independent of the program state, and *program* variables – non-rigid symbols that are program state dependent.

The verification of Java programs in the KeY system is based on symbolic execution realised through a sequent calculus. Program p in the modality is transformed by dedicated calculus rules to progressively reduce the program p into a description of the resulting program state. This description, denoted by $\mathcal{U}$, is called an update, and is essentially a set of canonical assignments of terms to program variables. The following two rules are characteristic for JDL:

$$\frac{\Gamma, \mathcal{U}\mathtt{b} \models \mathcal{U}\langle\pi\, p\, \omega\rangle\phi \quad \Gamma, \mathcal{U}!\mathtt{b} \models \mathcal{U}\langle\pi\, q\, \omega\rangle\phi}{\Gamma \models \mathcal{U}\langle\pi\, \mathtt{if(b)\{}p\mathtt{\}else\{}q\mathtt{\}}\, \omega\rangle\phi}\ \mathsf{if} \qquad \frac{\Gamma \models \mathcal{U}\{\mathtt{v} := \mathtt{se}\}\langle\pi\, \omega\rangle\phi}{\Gamma \models \mathcal{U}\langle\pi\, \mathtt{v=se;}\, \omega\rangle\phi}\ \mathsf{assign}$$

In these rules π denotes an inactive prefix of the program, e.g., a **`try{`** block opening, a label, or a logic-only description of the current method call stack. The remaining statements of the verified program that the current rule does not operate on are denoted with ω. The if rule unfolds the **if**-statement, which is removed from the modality, and two proof branches are created, where the execution of the two **if** branches, resp. p and q, can continue. The branch condition `b` is evaluated in the current state by applying the state update $\mathcal{U}$ to it. The assign rule transforms an assignment of a simple expression `se` to a local variable `v` into the update $\mathcal{U}$.

A complete symbolic execution of a program results in an empty modality and a set of updates that can be applied to the formula ϕ to check the validity of the initial claim $\langle p\rangle\phi$. If we consider $\mathcal{U}$ to be an operator on ϕ then it actually is another modality, one that only accepts sequences of canonical assignments as valid programs with the important property that the valuation of the formula ϕ can be quickly performed with a sequence of one-way update simplification and application rules, i.e., an equivalent of the weakest precondition calculus.

So far this covers only local Java variables, the need to reason about objects and arrays introduces the notion of a heap into the logic. The heap is represented as a dedicated program variable of a logic sort *Heap* and treated on equal grounds with other program variables. In particular, references and updates to the heap variable can directly appear in the set of updates $\mathcal{U}$. The *immutable* terms of the sort *Heap* are built using rigid function symbols *select* and *store*, that allow, respectively, querying the heap for a value of a given location, and constructing a new heap with some location updated to a new value w.r.t. some old heap. In particular, the assignment rule for updating an object field `f` is the following (fields are also first class citizens in JDL):

$$\frac{\Gamma \models \mathcal{U}\{\mathtt{heap} := \mathit{store}(\mathtt{heap}, \mathtt{o}, \mathtt{f}, \mathtt{se})\}\langle\pi\,\omega\rangle\phi}{\Gamma \models \mathcal{U}\langle\pi\,\mathsf{o.f{=}se;}\,\omega\rangle\phi}\ \mathsf{assignField}$$

For a specification language the KeY system employs JML*, an extension of JML [4] to accommodate dynamic frames [12]. This extension introduces the primitive type of location sets into JML and allows the assignable clauses to refer to variables of such a type instead of static locations. Since dynamic frames are an orthogonal issue to our formalisation of transactions, JML* is synonymous with JML for the work we present here. Moreover, only very basic JML constructs, that we assume the reader is familiar with, are discussed in the paper. In the verification process the KeY system translates a single Java method to be verified and the associated JML* specification into a JDL formula. In this process the heap variable is treated in a special way – it is the properties over this variable that need to be expressed to reflect any framing conditions specified in JML*. A very similar process is applied with similar implications on the heap variable when JML* specifications are used as axioms to replace method calls following the modular verification principles.

The JDL offers other strong facilities for reasoning about Java programs, e.g., the modelling of static initialisation, or comprehensive treatment of Java arithmetic including overflow. However, the work we present in this paper neither affects nor is affected by these other features of the logic. The KeY system itself is a GUI based user-friendly interactive verifier for JDL with a high degree of automation to minimise unnecessary interaction, often leading to fully automatic proofs even for considerably complex programs and properties.

2 Java Card Transactions on Explicit Heaps

In the following, driven by examples, we gradually present the complete formalisation of the Java Card transaction semantics in the KeY JDL and show how multiple heap variables are used. To start with, we introduce native transaction statements to the Java syntax handled by the logic. That is, the logic should allow for the symbolic execution of `#beginTr`, `#commitTr`, and `#abortTr` that define the transaction boundaries in the verified program. Bridging the actual transaction calls from the API to these statements is a straightforward extension of the verification system. Then, consider the snapshot (slightly artificial on purpose) of a Java Card program on the right, where the fields `balance` and `opCount` of object **this** are persistent, permanently storing the current balance and operation count of some payment application. The local variables `change` and `newBalance` are transient. Ignoring the transaction statements for the moment, the symbolic execution of this program results in the following state updates:

```
int newBalance = 0;
#beginTr;
this.opCount++;
newBalance =
  this.balance + change;
if(newBalance < 0) {
  #abortTr;
}else{
  this.balance = newBalance;
  #commitTr;
}
```

$$\begin{aligned}
&\mathtt{newBalance} := 0,\\
&\mathtt{heap} := \mathit{store}(\mathtt{heap}, \mathbf{this}, \mathtt{opCount}, \mathit{select}(\mathtt{heap}, \mathbf{this}, \mathtt{opCount}) + 1),\\
&\mathtt{newBalance} := \mathit{select}(\mathtt{heap}, \mathbf{this}, \mathtt{balance}) + \mathtt{change},\\
&\mathtt{heap} := \mathit{store}(\mathtt{heap}, \mathbf{this}, \mathtt{balance}, \mathtt{newBalance}) \quad (\text{when } \mathtt{newBalance} \geq 0)
\end{aligned}$$

The symbolic execution of the **if** statement splits the proof, so the last update only appears on the **else** proof branch where `newBalance` ≥ 0 is assumed.

After further simplification, this set of state updates can be applied to evaluate a property querying e.g., the value of operation count, which in the logic would be the term $\mathit{select}(\mathtt{heap}, \mathbf{this}, \mathtt{opCount})$. The result would indicate a one unit increase w.r.t. the value stored on the heap before this code is executed.

Basic Transaction Roll-Back. Assuming a simplified Java Card definition, updates to local variables should be kept, while the updates to persistent locations should be rolled back to the state before the transaction was started. The persistent locations in the actual program are synonymous with the data stored on the heap in the logic. Hence, in the first attempt it should be sufficient to roll back the value of the whole heap. This can be done by introducing two simple rules for transaction statements `#beginTr` and `#abortTr` that, respectively, store and restore the value of the heap to and from a backup heap variable `bHeap`:

$$\frac{\Gamma \models \mathcal{U}\{\mathtt{bHeap} := \mathtt{heap}\}\langle\pi\ \omega\rangle\phi}{\Gamma \models \mathcal{U}\langle\pi\ \mathtt{\#beginTr};\ \omega\rangle\phi}\ \mathsf{begin} \qquad \frac{\Gamma \models \mathcal{U}\{\mathtt{heap} := \mathtt{bHeap}\}\langle\pi\ \omega\rangle\phi}{\Gamma \models \mathcal{U}\langle\pi\ \mathtt{\#abortTr};\ \omega\rangle\phi}\ \mathsf{abort}$$

This can be done and works as expected because the `heap` variable as modelled in KeY JDL has *call by value* characteristics. Now the set of state updates (on the negative `newBalance` branch) of our example program is the following:

$$\begin{aligned}
&\mathtt{newBalance} := 0,\\
&\mathtt{bHeap} := \mathtt{heap},\\
&\mathtt{heap} := \mathit{store}(\mathtt{heap}, \mathbf{this}, \mathtt{opCount}, \mathit{select}(\mathtt{heap}, \mathbf{this}, \mathtt{opCount}) + 1),\\
&\mathtt{newBalance} := \mathit{select}(\mathtt{heap}, \mathbf{this}, \mathtt{balance}) + \mathtt{change},\\
&\mathtt{heap} := \mathtt{bHeap} \quad (\text{when } \mathtt{newBalance} < 0)
\end{aligned}$$

Whatever terms referring to heap contents should be evaluated with this set of updates, the result would be the values on the heap at the point where it was saved in the `bHeap` variable. The commit statement needs no special handling apart from silent stepping over this statement. In this case the saved value of the `heap` in the `bHeap` variable is simply forgotten until a possible subsequent new transaction where `bHeap` is freshly overwritten with a more recent `heap`.

For the very superficial treatment of transaction semantics this is enough to model transactions in JDL. Note that, so far, no new or assignment rules of any kind were introduced and the new heap variable `bHeap` is not modified in any way apart from being initialised to hold a complete copy of the regular heap.

Transaction Marking and Balancing. The two rules we just introduced do not enforce any order on the transaction statements, they allow to successfully verify malformed programs like "`#abortTr; #beginTr;`" or "`#commitTr; #commitTr;`". Furthermore, by Java Card specification, transactions cannot be nested, i.e., the maximum allowed transaction depth is 1, attempts to exceed this limit cause a run-time exception. On the other hand, the scope of a single transaction is very liberal according to the specification – a transaction can be in progress for as long as the card session is active, regardless of the stack of method calls. To simplify our formalisation, we opt for enforcing a stronger requirement – a transaction should be contained in one single Java method. That is, any method that opens a transaction has to close it before the method terminates. This does not exclude complete methods to be called during a transaction, but it does exclude a transaction opening in one method, and closing in another one that is eventually called later on. Our requirement is justified by Java Card security guidelines [21] that ban programs with transaction blocks spanning over several methods (to prevent transaction buffer overruns).[3] In practice, our formalisation not only relies on this requirement, but also enforces it, i.e., programs not adhering to this requirement do not verify.

Consequently, our formalisation restricts the transaction scope in the following way. A transaction marker *TR* attached to a modality indicates that the current execution context of the verified program is an open transaction. Rules for handling transaction opening and closing statements are now sensitive to this marker and automatically enforce correct transaction balancing. Similarly, rules for discharging empty modalities prevent closing proofs with a remaining transaction marker. In turn, any transaction block has to appear in a single verification context (modality), i.e., one method. Furthermore, the dedicated rule for array assignments can be singled out for transaction contexts only. This keeps verification of regular Java programs clear of any unnecessary transaction

[3] Following a similar security rationale we disallow object allocation inside transactions. Real Java Card programs cause serious security risks when objects are allocated in transactions [20], while the formalisation to deal with the "shady" semantics of object deallocation mandated by the Java Card specification [25] would require modelling of explicit garbage collection, something that Java verification systems in principle are not designed for.

artefacts in the proofs. Finally, knowing that the current point in the symbolic execution is a transaction context is important in modular verification for the local interpretation of heap parametric specifications as explained later in Sect. 3.

The rules for transaction statements are the following. An explicit rule for the commit statement is added, in which nothing happens to the `heap` variable, but the transaction context is cancelled out by removing the *TR* marker:

$$\frac{\Gamma \models \mathcal{U}\{\mathtt{bHeap} := \mathtt{heap}\}\langle_{TR}\pi\,\omega\rangle\phi}{\Gamma \models \mathcal{U}\langle\pi\;\#\mathtt{beginTr};\omega\rangle\phi}\;\textsf{begin}$$

$$\frac{\Gamma \models \mathcal{U}\{\mathtt{heap} := \mathtt{bHeap}\}\langle\pi\,\omega\rangle\phi}{\Gamma \models \mathcal{U}\langle_{TR}\pi\;\#\mathtt{abortTr};\omega\rangle\phi}\;\textsf{abort} \qquad \frac{\Gamma \models \mathcal{U}\langle\pi\,\omega\rangle\phi}{\Gamma \models \mathcal{U}\langle_{TR}\pi\;\#\mathtt{commitTr};\omega\rangle\phi}\;\textsf{commit}$$

Persistent and Transient Arrays. So far in our formalisation we roll back the whole contents of the backup heap, i.e., we operate the `bHeap` variable as a whole without changing single object locations on it. The separate transaction treatment for the persistent and transient arrays in Java Card now requires also selectively modifying the backup heap, as we describe in the following.[4]

The Java Card transaction rules require that the contents of transient arrays, allocated by dedicated API methods, are never rolled back. Since in JDL all arrays are stored on the heap, we somehow need to introduce a *selective* roll-back mechanism. We achieve this with the following. Whenever an array element is updated in a transaction we check for the persistency type of the array. The check itself is done by introducing an additional implicit boolean field to all objects, called `<transient>`, that maintains the information about the object's persistency type. Standard allocation rules set this field to false, while the dedicated API methods for creating transient arrays specify this field to be true.

Then, when handling assignments, for persistent arrays we take no additional action, for transient arrays we update the value on the heap and *simultaneously* update the value on the backup heap `bHeap`. During an abort, the regular heap is restored to the contents of the backup heap that now also includes updates to transient arrays that were not supposed to be rolled back. The core of the resulting assignment rule for arrays is the following:

$$\frac{\begin{array}{l}\Gamma, \mathcal{U}!\mathtt{a.<transient>} \models \mathcal{U}\{\mathtt{heap} := \mathit{store}(\mathtt{heap},\mathtt{a},\mathtt{i},\mathtt{se})\}\langle_{TR}\pi\,\omega\rangle\phi \\ \Gamma, \mathcal{U}\mathtt{a.<transient>} \models \mathcal{U}\{\mathtt{heap} := \mathit{store}(\mathtt{heap},\mathtt{a},\mathtt{i},\mathtt{se}), \\ \qquad\qquad\qquad\qquad \mathtt{bHeap} := \mathit{store}(\mathtt{bHeap},\mathtt{a},\mathtt{i},\mathtt{se})\}\langle_{TR}\pi\,\omega\rangle\phi\end{array}}{\Gamma \models \mathcal{U}\langle_{TR}\pi\;\mathtt{a[i]} = \mathtt{se};\omega\rangle\phi}\;\textsf{arrayAssign}$$

Assuming that arrays `tr` and `ps` are, respectively, transient and persistent, the symbolic execution of this program:

```
tr[0] = ps[0] = 0; #beginTr; tr[0] = 1; ps[0] = 1; #abortTr;
```

results in the following sequence of state updates:

[4] Only arrays can be made persistent or transient in Java Card, regular objects are always persistent. Thus, we only discuss arrays in this context, but our formalisation works for regular objects, too.

$$\begin{aligned}&\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{tr}, 0, 0),\ \texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{ps}, 0, 0),\\ &\texttt{bHeap} := \texttt{heap},\\ &\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{tr}, 0, 1),\ \texttt{bHeap} := \mathit{store}(\texttt{bHeap}, \texttt{tr}, 0, 1),\\ &\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{ps}, 0, 1),\\ &\texttt{heap} := \texttt{bHeap}\end{aligned}$$

With these updates, the valuation of $\mathit{select}(\texttt{heap}, \texttt{ps}, 0)$ and $\mathit{select}(\texttt{heap}, \texttt{tr}, 0)$ results in resp. 0 and 1 as required by the Java Card transaction semantics.

Non-atomic Updates. The last complication in the transaction rules are the so-called *non-atomic* updates of persistent array elements. Such updates bypass transaction handling, i.e., no roll-back of data updated non-atomically is performed. Updates to transient arrays as defined by Java Card are in fact non-atomic, as they are never rolled back either. We have just introduced a mechanism that prevents the roll-back of transient arrays, by checking the `<transient>` field of the array and providing corresponding state updates. To extend this behaviour to persistent arrays, we allow for the implicit `<transient>` field of an array to be mutable in our logic. In turn, we can temporarily change the assignment semantics for an array by manipulating the `<transient>` field. Concretely, a non-atomic assignment to a persistent array element can be modelled by first setting the `<transient>` field to true, then performing the actual assignment, and then changing the value of `<transient>` back to false. Hence, a non-atomic assignment "a[i] = se;" to a persistent array a, is simply modelled as:

```
a.<transient> = true; a[i] = se; a.<transient> = false;
```

Then, the array assignment rule we provided above introduces the necessary updates to the regular and backup heaps to achieve transaction bypass.

In Java Card the non-atomic updates are delegated to dedicated API methods, i.e., they are not part of the language syntax. Hence, the manipulation of the `<transient>` field is delegated to the reference implementation of these API methods, and this *emulation* of non-atomic assignments is easily achieved in the actual Java Card programs to be verified by KeY.

```
a[0] = 0;
#beginTr;
 a[0] #= 1; a[0] = 2;
#abortTr;
```

```
a[0] = 0;
#beginTr;
 a[0] = 2; a[0] #= 1;
#abortTr;
```

Unfortunately, there is one more condition for non-atomic updates that we need to check. A request for a non-atomic update becomes effective only if the persistent array in question has not been already updated atomically (i.e., with a regular assignment) within the same transaction. If such an update has been performed, any subsequent updates to the array are always atomic within the same transaction and rolled back upon transaction abort. We illustrate this with two simple programs operating on a persistent array a above on the right, for simplicity we mark non-atomic assignments with #= instead of quoting the actual API call that does that. The top program results in a[0] equal to 1 (a non-atomic update is in effect), the bottom program rolls a[0] back to 0, as the regular assignment "a[0] = 2;" disables any subsequent non-atomic assignments, and hence all transaction updates are reverted.

To introduce this additional check in the logic, we employ one more implicit field for array objects, `<trUpdated>`, that maintains information about atomic updates. Set to true, it indicates that the array was already updated with a regular assignment, false indicates no such updates and allows for non-atomic updates in the same transaction still to be effective. The new assignment rule for arrays needs to be altered to handle all these conditions and also to record the changes to the `<trUpdated>` field itself. The saturated state updates to be introduced under different conditions in the assignment rule for "`a[i] = se;`" are the following:

Condition	State update
Always	$\mathtt{heap} := \mathit{store}(\mathtt{heap}, \mathtt{a}, \mathtt{i}, \mathtt{se})$
`!a.<transient>`	$\mathtt{bHeap} := \mathit{store}(\mathtt{bHeap}, \mathtt{a}, \mathtt{<trUpdated>}, \mathit{TRUE})$
`a.<transient>` and `!a.<trUpdated>`	$\mathtt{bHeap} := \mathit{store}(\mathtt{bHeap}, \mathtt{a}, \mathtt{i}, \mathtt{se})$

The updates to the `<trUpdated>` field are purposely stored on the backup heap to ease the resetting of this field with each new transaction, because the backup heap is freshly assigned with each new transaction while the regular heap is not. Now, on transaction abort, the heap reverting update filters out any updates to this field on the backup heap using the anonymisation function of the JDL:

$$\mathtt{heap} := \mathit{anon}(\mathtt{bHeap}, \mathit{allObjects}(\mathtt{<trUpdated>}), \mathtt{heap})$$

Intuitively, this expresses the operation of copying the contents of heap `bHeap` to `heap`, but retaining the value of the `<trUpdated>` field in all objects in `heap`. This way all manipulations of `<trUpdated>` in proofs are local to a single transaction.

3 Heaps as Parameters in JML*

The previous section spelled out the details of formalising Java Card transactions in JDL. The key point in this formalisation is the modified assignment rule in the sequent calculus that now operates on two heap variables. In some sense, assignment rules are always the core of the program logic – they give semantics of state changes for the verified program. A specification language that describes the program behaviour also deals in a large part with the corresponding state changes (or lack thereof). Hence, one can say there is a special correspondence between the assignment rules and the specification language.

This means that specifications for methods called in transactions should additionally express properties about data on the backup heap together with the framing conditions. To this end we introduced the following extensions. To redirect any object field access `o.f` to a different heap one can use the **`\at`** operator, e.g., accessing data on the backup heap is expressed with `\at(backupHeap,o.f)`. In this context, the plain field access `o.f` in fact means `\at(heap,o.f)`. Then, for framing specifications, the assignable clauses also take a heap parameter to bind locations with a corresponding heap:

```
assignable[heap] o.f;
assignable[backupHeap] o.g;
```

```
/*@ public normal_behavior
      requires len >= 0 && off >= 0 && off + len <= a.length;
      ensures \result == off + len;
      ensures (\forall int i; i>=0 && i<len; a[off + i] == v);
      requires[backupHeap] JCSystem.getTransactionDepth() == 1;
      requires[backupHeap] a.<transient> ==> !a.<trUpdated>;
      ensures[backupHeap] (\forall int i; i>=0 && i<len;
        \at(backupHeap, a[off + i]) ==
            ((!a.<transient> && \at(backupHeap, a.<trUpdated>) ?
                 \old(\at(backupHeap, a[off + i])) : v) );
      assignable[heap,backupHeap] a[off..off+len-1]; @*/
public static int arrayFillNonAtomic(byte[] a, int off, int len, byte v);
```

Fig. 1. Complete JML* specification for one of the Java Card API methods

Now it is possible to generate separate proof obligations for the framing conditions for the two heaps and correctly apply method contracts in the presence of two heaps.

We generalise this further. Any specification element in JML*, like a precondition specified with the **requires** clause, receives a heap parameter. This parameter specifies the applicability context of the given specification element. In particular, specification elements defined for the backup heap are only considered in verification contexts of an open transaction, i.e., within the marked $\langle_{TR}\cdot\rangle$ modality. Specification elements not annotated with any heap apply to the default heap that is always active. This way we achieve transparency – old style specifications refer to the regular heap by default and retain their previous semantics. An illustration for this is given in Fig. 1, where a complete specification for the Java Card API method for updating chunks of arrays in a non-atomic way is given for both the transaction and non-transaction contexts.

4 Implementation in KeY

Implementing the support for Java Card transactions in KeY was done in two steps. The first step was to generalise the JML* interface to accept multiple heaps and convey the information about them to the proof obligation generation component and the modular reasoning component. This was simply done by considering an arbitrary list of heaps in the corresponding modules rather than referring to the one predefined heap. Until this point the extensions were fully generic, i.e., not specific to the Java Card transaction mechanism in any way. In particular, the generation of concrete formulae for framing conditions remained the same, only now several ones for different heaps are created.

In the second step we added the core formalisation of Java Card transactions to the KeY system. In KeY the logic rules are defined externally, using the so-called taclet language [2, Chap. 4] for defining the corresponding rewrites. The *TR* marker was added by simply declaring a new modality. Then a handful of new rules we discussed in Sect. 2 were added to the rule base. As an example, a self-explanatory taclet for the `#beginTr` statement is the following in KeY:

```
beginJavaCardTransaction {
  \find (==> \diamond{.. #beginTr; ...}\endmodality phi)
  \replacewith(==> {backupHeap := heap}
      \diamond_transaction{.. ...}\endmodality phi) };
```

Apart from this rule and the transaction specific rule for array assignments the addition of the second heap variable `backupHeap` required only declaring it. This declaration automatically tells the other components of the KeY system to include it (considering the current verification context) in the corresponding verification tasks, like proof obligation generation or modular application of contracts.

To evaluate our work we revisited our earlier work on the fully verified reference implementation of the Java Card API [19]. We specified the Java Card API methods following the extended JML* syntax (see again Fig. 1) and verified both the reference implementation of the API as well as a handful of other Java Card examples that make calls to the Java Card API.

The overall result of our work shows considerable improvements compared to our old formalisation of transactions [3,18] back when the heap model in JDL was not based on explicit heap access through a special program variable. The complete set of changes to the logic and the calculus is now much smaller, the implementation overhead of the new rules practically negligible, and finally the resulting automatic proofs for Java Card programs much more readable. We attribute these improvements to the use of multiple heaps, which was not possible before. Previously, the semantics of state updates on the implicit heap had to be heavily modified to include a notion of a *forgetting* update to model data rollback in the logic with deep implications for the calculus and the implementation. Preliminary work in the area of concurrent verification provides another strong case for the explicit use of multiple heaps as we briefly describe next.

5 New Applications for Multiple Heaps in Verification

In the introduction we mentioned distributed computing as an example where multiple heaps should be considered in the computation model, with at least the local and one remote heap. Another scenario is low level reasoning about systems with (possibly multi-level) cache memory, where one heap would represent the cache and one the main memory. Here the verification could concentrate on the data dependencies and synchronisation between the cache and the main memory. Going further, multi-core systems (like GPUs) could be also modelled using multiple explicit heaps, each heap representing the local memory of a single core. Finally, the real-time Java can be also considered in this context, where programs access memories with different physical characteristics on one embedded device [13].

In the context of the ongoing VerCors project[5] [1] we currently concentrate on extending the KeY logic to deal with permission based verification of concurrent programs. Permission accounting is a specification oriented methodology for ensuring race freedom in concurrent programs that allows for efficient thread-local reasoning. Similarly to the implementation of permissions in the Chalice tool [15,22] we introduce a permission mask to the JDL to keep track of permissions in the verified programs. From our point of view, this permission mask is nothing more than a parallel heap-like structure that stores permission values for each location instead of the actual values. In the first experimental attempt, using the multiple heap framework that we discussed, we simply added a new heap structure to the logic, represented with the program variable `permissions`, to keep track of the permissions that the local Java thread owns. The location assignment and access rules were amended to ensure, respectively, a write or read permission to a given location. Now, using our heap-aware JML*, we can give permission based specifications:

```
requires[permissions] \at(permissions, o.f) == 1;
assignable o.f;
assignable[permissions] o.f;
```

This states that we require a write permission to the location `o.f`, that this location is changed on the actual heap (the regular **assignable**), and also that the permission to the location may be modified, e.g., through permission transfer to another thread. Disregarding any specification clauses associated with permissions, in the example the first and the third line, transforms the specification into a permission unaware specification. This can be useful for verifying permission and functional properties separately. Very basic examples with permissions have been already verified with an experimental version of KeY.

6 Conclusions

In this paper we discussed the use of multiple heaps in formal verification of Java programs using the formalisation of Java Card atomic transactions fully implemented in KeY as an example. We also took the opportunity to give full details of this formalisation that were not yet published elsewhere. In the ongoing work we apply the same methodology to introduce permission based reasoning for concurrent Java programs in KeY. Few other applications in verification have been named as possible directions for more future work.

It seems that none of the other verification systems that we are aware of try to make heap or heap-like structures explicit on the level of the specification language, although certainly some of them indeed use multiple heap or heap-like structures internally. Most notably, the Chalice tool [15,22] works with two global variables H and P, that, respectively, represent the heap and the permission mask in the Boogie proof obligations. Not exposing the heap in the Separation Logic specifications and associated tools [11,7] seems natural, however, applying

[5] `http://fmt.cs.utwente.nl/research/projects/VerCors/`

them to new verification scenarios named in Sect. 5 becomes significantly more difficult in our opinion.

When it comes to the formalisation of Java Card atomic transactions, only the Krakatoa tool [17] also provides a sound formalisation and implementation of the transaction roll-back that accounts for the specifics of non-atomic methods. The Krakatoa formalisation relies on keeping extra copies of data to be rolled back on the same heap as all the other data in dedicated backup fields associated with regular fields, i.e., all data fields are backed-up separately instead of the whole heap. This is very similar to our first formalisation of transactions [3], which turns out to be very heavy-weight compared to our current work. We believe that our current formalisation can be applied easily in other verification systems, as long as such a system is capable of manipulating the heap variable as we do in the KeY logic. A partial support for Java Card transactions has been also recently reported for the VeriFast platform [10], however, the semantics of the transaction roll-back has not been formalised there. Finally, Java Card transactions have been considered to be formalised in the LOOP tool using program transformation to explicitly model transaction recovery directly in the Java code, but the ideas where never implemented in the tool [9].

Acknowledgements. The work of W. Mostowski is supported by ERC grant 258405 for the VerCors project. We would like to thank Richard Bubel for his insights and invaluable help with the implementation.

References

1. Amighi, A., Blom, S.C., Huisman, M., Zaharieva-Stojanovski, M.: The VerCors project: Setting up basecamp. In: 6th Workshop Programming Languages Meets Program Verification, pp. 71–82. ACM (2012)
2. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
3. Beckert, B., Mostowski, W.: A program logic for handling Java Card transaction mechanism. In: Pezzé, M. (ed.) FASE 2003. LNCS, vol. 2621, pp. 246–260. Springer, Heidelberg (2003)
4. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. In: Arts, T., Fokkink, W. (eds.) 8th Int'l Workshop on Formal Methods for Industrial Critical Systems. ENTCS, vol. 80, pp. 73–89. Elsevier (2003)
5. Burdy, L., Requet, A., Lanet, J.-L.: Java applet correctness: A developer-oriented approach. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 422–439. Springer, Heidelberg (2003)
6. Chen, Z.: Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison-Wesley (June 2000)
7. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. SIGPLAN Notes 43, 213–226 (2008)
8. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
9. Hubbers, E., Poll, E.: Reasoning about card tears and transactions in Java Card. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 114–128. Springer, Heidelberg (2004)

10. Jacobs, B., Smans, J., Philippaerts, P., Piessens, F.: The VeriFast program verifier – a tutorial for Java Card developers. Technical report, Department of Computer Science, Katholieke Universiteit Leuven, Belgium (September 2011)
11. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011)
12. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
13. Kwon, J., Wellings, A.J.: Memory management based on method invocation in RTSJ. In: Meersman, R., Tari, Z., Corsaro, A. (eds.) OTM-WS 2004. LNCS, vol. 3292, pp. 333–345. Springer, Heidelberg (2004)
14. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
15. Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with Chalice. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) Foundations of Security Analysis and Design, pp. 195–222. Springer (2009)
16. Marché, C., Paulin-Mohring, C., Urbain, X.: The Krakatoa tool for certification of Java/Java Card programs annotated in JML. Journal of Logic and Algebraic Programming 58(1-2), 89–106 (2004)
17. Marché, C., Rousset, N.: Verification of Java Card applets behavior with respect to transactions and card tears. In: Hung, D.V., Pandya, P. (eds.) 4th IEEE Conference on Software Engineering and Formal Methods. IEEE Press (2006)
18. Mostowski, W.: Formal reasoning about non-atomic Java Card methods in dynamic logic. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 444–459. Springer, Heidelberg (2006)
19. Mostowski, W.: Fully verified Java Card API reference implementation. In: Beckert, B. (ed.) 4th Int'l Verification Workshop. CEUR WS, vol. 259 (2007)
20. Mostowski, W., Poll, E.: Malicious code on java card smartcards: Attacks and countermeasures. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 1–16. Springer, Heidelberg (2008)
21. Pallec, P.L., Saif, A., Briot, O., Bensimon, M., Devisme, J., Eznack, M.: NFC cardlet development guidelines v2.2. Technical report, Association Française du Sans Contact Mobile (2012)
22. Parkinson, M.J., Summers, A.J.: The relationship between separation logic and implicit dynamic frames. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 439–458. Springer, Heidelberg (2011)
23. Schmitt, P.H., Ulbrich, M., Weiß, B.: Dynamic frames in Java dynamic logic. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 138–152. Springer, Heidelberg (2011)
24. Stenzel, K.: A formally verified calculus for full Java Card. In: Rattray, C., Maharaj, S., Shankland, C. (eds.) AMAST 2004. LNCS, vol. 3116, pp. 491–505. Springer, Heidelberg (2004)
25. Sun Microsystems, Inc., Java Card 2.2.2 Runtime Environment Specification (March 2006), `http://www.oracle.com`

Parameterized Verification of Track Topology Aggregation Protocols

Sergio Feo-Arenis and Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Abstract. We present an approach for the verification aggregation protocols, which may be used to perform critical tasks and thus should be verified. We formalize the class of track topology aggregation protocols and provide a parameterized proof of correctness where the problem is reduced to checking a property of the node's aggregation algorithm. We provide a verification rule based on our property and illustrate the approach by verifying a non-trivial aggregation protocol.

1 Introduction

We study the verification of *aggregation protocols*, which are often used in sensor networks [7,12,13]. Such protocols compute an aggregation function in a distributed fashion. An aggregation function is a function of the data at sensor nodes where the messages sent by a node are determined by messages received from other nodes and the own data. Data is propagated from sensor nodes towards specially designated sink nodes, where the result of the computation is made available.

A simple example of an aggregation protocol is one where the maximum value among the sensors' data is calculated at a sink node. Here, if intermediate nodes transmit only the maximum value among the received data as it is forwarded to the sink, the utilization of resources can be improved. The reduced amount of data packets can, e.g., reduce energy usage, reduce the use of available bandwidth and increase the maximum data collection rate.

Sensor networks may use unreliable communication channels. In order to mitigate the problem of unreliable communications, sensor networks may take advantage of the fact that broadcast transmissions may be received by multiple nodes. In this way, they effectively provide multiple *aggregation paths* over which a node's data can be collected.

Some aggregation functions are, however, *duplicate-sensitive*. That is, obtaining a correct result requires that sensor values are aggregated only once. This is the case when, e.g., calculating the sum or the average of the sensor values.

Data aggregation protocols are correct in the presence of unreliable communications if, whenever an aggregation path is available from a node to the sink, the sensor's data is correctly aggregated at the sink. I.e., it is aggregated, but exactly once if the function being calculated is duplicate-sensitive. This concept of correctness is the strongest possible when unreliable communications are employed. The failure of all aggregation paths cannot be completely ruled out.

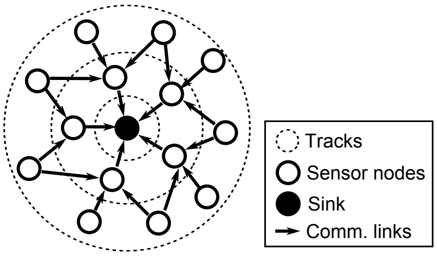


Fig. 1. Track Topology

Our goal is to provide a method to (semi-)automatically verify correctness of aggregation protocols given the implementation of the algorithm executed by network nodes. As a first approach, we concentrate on *track topologies*, where nodes are arranged in *tracks* according to their distance (in hops) to the sink, and aggregation paths traverse consecutive track boundaries as shown in Figure 1.

We study aggregation protocols for topologies that have a reliable *correction infrastructure*. I.e., nodes within tracks have reliable communication links that can be used to correct communication errors. This is the case for, e.g., the use of short-range directional antennas or aggregation systems where communication units have multiple transceivers.

We additionally assume a *schedule*, i.e., the sequence in which nodes perform their computations and transmit their data, such that the data from input nodes is always available at the time of executing a node's algorithm.

Even under those assumptions, verifying correctness of an aggregation protocol requires considering all possible topologies, with a possibly unbounded number of nodes. Aggregation protocols are parameterized systems and can thus not be verified by explicitly verifying all instances. The problem of parameterized verification is undecidable in general [1].

We approach the problem of verifying a specific class of aggregation protocols in this work. We make the following contributions.

1. We identify the class of aggregation protocols for track topologies with reliable correction infrastructure and provide a formalization as a parameterized verification problem, including the presence of link failures.
2. We provide a proof rule that reduces the parameterized verification problem to checking a verification condition on the nodes' algorithm and show its soundness. We thus provide an inductive invariant for the verification of track topology aggregation protocols.
3. We illustrate the utility of our results by applying the verification rule to a non-trivial aggregation protocol.

This work illustrates the possibility of performing parameterized verification for distributed systems with topologies that are not regular and with dynamic aspects such as node or link failures. We are able to concentrate the effort of verification on single nodes, without the need to consider topology or scheduling issues. Our formalization could be used together with approaches such as [14] to obtain parametric proofs for such systems. Formalizing link and node reliability can provide the basis for deriving inductive invariants.

2 Track Topologies and Aggregation Protocols

We define the class of systems that are the subject of our verification effort. Inspired by instances seen in wireless sensor networks [7], we develop the concept of track topologies by establishing the properties that characterize them.

Definition 1. *A* track topology

$$T = (N, \longrightarrow, \rightsquigarrow, V)$$

with domain $\mathcal{D}$ is a finite, node-labeled, acyclic graph with nodes $N = \{n_1, \ldots, n_k\}$, edges $\longrightarrow \cup \rightsquigarrow \subseteq N \times N$, and labeling function

$$V : N \to \mathcal{D}$$

which satisfies the following conditions:

- *The set of nodes is partitioned into $d \in \mathbb{N}_0$ non-empty sets $N_0, \ldots, N_d$ called* tracks. *We call N_0 the* sink track *and use $track(n)$ to denote the number of the track to which node $n \in N$ belongs.*
  The number d is called the depth *of the topology, denoted by $depth(T)$.*
- *The relation $\longrightarrow$ is called* aggregation relation *and connects nodes in a track with nodes in the track immediately before, i.e.,*

$$\forall\, (n, n') \in \longrightarrow\ \bullet\ track(n) = track(n') + 1 \tag{1}$$

  For $(n, n') \in \longrightarrow$, n' is called aggregator *for n. A path from a node n to a node $n' \in N_0$ is called* aggregation path.
- *The relation $\longrightarrow$ consists of two disjoint relations, the* primary aggregation relation *PA and the* secondary aggregation relation *SA, such that (N, PA) constitutes a forest and (N, SA) is a directed, acyclic graph.*
  For $(n, n') \in PA$ (SA), n' is called primary parent *(*secondary parent*) of n. We introduce the abbreviation $SA(n) = \{n' \mid (n', n) \in SA\}$ to denote the set of* secondary children *of n.*
- *The relation $\rightsquigarrow$ is disjoint from $\longrightarrow$ and called* correction relation*. It connects the primary parent of a node with all of its backup parents, i.e.*

$$\begin{aligned} &\forall\, (n, n') \in PA\ \exists\, n_0, n_1, \ldots, n_m \bullet \\ &\qquad n_0 = n' \wedge \forall\, 0 \le i < m \bullet n_i \rightsquigarrow n_{i+1} \wedge SA(n) \subseteq \{n_0, \ldots, n_m\}. \end{aligned} \tag{2}$$

  Note that due to the restriction that the aggregation relation connects only adjacent tracks, $\rightsquigarrow$ always connects nodes in the same track.

k is called the size *of T, denoted by $|T|$.*

An example of a track topology with primary, secondary and correction relations can be seen in Figure 2.

For our inductive proof, we require track topologies to be closed under removal of single nodes in the sink track. By allowing to remove only nodes in the sink track without outgoing correction edges, it is ensured that no correction path between a primary aggregator and the backup aggregators of some node is broken, thus preserving the property (2).

Lemma 1. *Let $n_0 \in N_0$ be a node from the sink track of a track topology $T = (N_0, \longrightarrow_0, \rightsquigarrow_0, V_0)$ with domain $\mathcal{D}$ which does not have any outgoing edges in the correction relation to any other node.*

Then $T \setminus \{n_0\} := (N, \longrightarrow, \rightsquigarrow, V)$ with

- *$N = N_0 \setminus \{n_0\}$, $V = V_0|_N$,*
- *$\longrightarrow = \{(n, n') \in \longrightarrow_0 \mid n' \neq n_0\}$, $\rightsquigarrow = \{(n, n') \in \rightsquigarrow_0 \mid n' \neq n_0\}$,*
- *$track(n) = track_0(n) - 1$ if $N_{0_0} = \{n_0\}$, and $track(n) = track_0(n)$ otherwise,*

is a track topology with domain $\mathcal{D}$.

Whenever aggregation is performed, nodes initiate a transmission round, each transmitting their computed messages over the communication medium. In the following, we formalize unreliable communication links. We introduce an edge labeling function for the aggregation relation of a topology that indicates whether communication is successful or not during a transmission round.

Definition 2. *A* communication function *of a track topology $T = (N, \longrightarrow, \rightsquigarrow, V)$ is an edge labeling*

$$f : (\longrightarrow) \to \mathbb{B}$$

of the aggregation relation $\longrightarrow$ of T where $f(n, n') = 1$ if and only if the communication was successful between the connected nodes n and n'. We use F_T to denote the set of communication functions of T, and $n \longrightarrow_f n'$ if and only if $f(n, n') = 1$.

The value of node n is available at the sink track *if and only if there exists a $\longrightarrow$-path from n to a node in the sink track such that the communication along the path was successful according to f. We use $V_f(n)$ to denote the value of n if the value is available at the sink track according to f and "$\circ$" otherwise, i.e. V_f is pointwise defined as*

$$V_f(n) = \begin{cases} V(n) & , \textit{if} \quad \exists\, n_0 \in N_0 \bullet n \longrightarrow_f^* n_0, \\ \circ & , \textit{otherwise.} \end{cases} \tag{3}$$

In the following, we present the formalization of aggregation protocols and their correctness, which provides the framework necessary to develop a formal verification approach.

We observe that an aggregation protocol can be formalized as a function that maps a set of sensor values to a final aggregation value. For simplicity, and without loss of generality, we assume that the domain of the sensor values and that of the aggregation result are the same. In practice, complex functions may utilize different domains.

We continue by defining correctness of an aggregation protocol in the presence of unreliable communication links. For that, we formalize correctness relative to the communication function f: whenever there is a working path from a node to a sink, the data of the node should arrive (in aggregated form) at the sink.

The assumption that the correction infrastructure is reliable plays an important role. If the correction links were not reliable, then one would have to define correctness not only for working aggregation paths but also for working correction paths. We restrict our analysis to a reliable correction infrastructure. Extending it to unreliable correction links should work in a similar fashion and is the subject of future work.

Definition 3. *Let $\mathcal{T}$ be a set of track topologies with domain $\mathcal{D}$ and*

$$\mathcal{P} : \mathcal{T} \times F_{\mathcal{T}} \to \mathcal{D},$$

a track topology aggregation protocol *on $\mathcal{T}$, where F_T denotes the set of communication functions of the track topologies in $\mathcal{T}$.*

Let $\mathcal{A}$ be a family of aggregation functions

$$\mathcal{A}^k : \mathcal{D}_\circ^k \to \mathcal{D},$$

over domain $\mathcal{D}$ where $k \in \mathbb{N}_0$ and $\mathcal{D}_\circ = \mathcal{D} \sqcup \{\circ\}$.

The aggregation protocol $\mathcal{P}$ is called correct *wrt. $\mathcal{A}$ if and only if, whenever the value of a node n is available to the sink track, then the data transmitted by n is aggregated exactly once by $\mathcal{P}$, i.e. if*

$$\begin{aligned} &\forall T = (N, \longrightarrow, \rightsquigarrow, V) \in \mathcal{T}, k = |N| \; \forall f \in F_T \bullet \\ &\mathcal{P}(T, f) = \mathcal{A}^k \left(V_f(n_1), \ldots, V_f(n_k)\right). \end{aligned} \tag{4}$$

Now we formally characterize the aggregation protocols that are the subject of this work. We formalize the distributed calculation of the aggregation function as an equation system over local aggregation functions that map incoming messages and additional parameters to outgoing messages. The most prominent requirement in our formalization is that correction requests are directly observable in the network transmissions, thus allowing reasoning about the consistency of the messages computed by the nodes.

Definition 4. *Let $\mathcal{T}$ be a set of track topologies with a domain $\mathcal{D}$ such that $(\mathcal{D}_\circ, \oplus, \circ)$ is a monoid. A track topology aggregation protocol $\mathcal{P}$ on $\mathcal{T}$ is called* distributed homogeneous *if and only if*

– *there exists a family of sets Msg_k of* messages *together with two families*

$$\mathscr{A}^k : Msg_k^k \to \mathcal{D}$$
$$\mathscr{E}^k : Msg_k^k \to 2^{ID_k}$$

of decoder functions, $k \in \mathbb{N}_0$, $ID_k = \{1, \ldots, k\}$.
Decoder function $\mathscr{A}^k$ *obtains the aggregation value encoded in a message and* $\mathscr{E}^k : Msg_k^k \to 2^{ID_k}$ *extracts from a message the* correction requests *contained in that message, i.e., the set of nodes encoded in the message for which a corrective action should be taken if possible,*

– *there exists a family of* local aggregation functions

$$aggr^k : ID_k \times \mathcal{D} \times (ID_k \nrightarrow Msg_k) \to Msg_k, k \in \mathbb{N}_0,$$

such that, for track topology $T = (N, \longrightarrow, \rightsquigarrow, V) \in \mathcal{T}$ *and communication function* $f \in F_T$, *the aggregation result of the protocol* $\mathcal{P}$ *is the* $\oplus$*-sum of the messages emitted by the sink track, i.e.*

$$\mathcal{P}(T, f) = \bigoplus_{n_i \in N_0} \mathscr{A}(\tau_i) \tag{5}$$

where message τ_i *of node* $n_i \in N$ *in the track topology is defined by applying the local aggregation function to the identity* i *of the node, its value, and the set of messages relevant for* n_i *which have successfully been communicated according to* f.
That is, the messages $\tau_1, \ldots, \tau_k$ *are the solution of the equation system*

$$\begin{aligned} \tau_1 &= aggr^k(1, Vn_1, M_1) \qquad & M_1 &= \{i \mapsto \tau_i \mid (n_i, n_1) \in (\longrightarrow_f \cup \rightsquigarrow)\} \\ \tau_2 &= aggr^k(2, Vn_2, M_2) \qquad & M_2 &= \{i \mapsto \tau_i \mid (n_i, n_2) \in (\longrightarrow_f \cup \rightsquigarrow)\} \\ &\vdots & &\vdots \\ \tau_k &= aggr^k(k, Vn_k, M_k) \qquad & M_k &= \{i \mapsto \tau_i \mid (n_i, n_k) \in (\longrightarrow_f \cup \rightsquigarrow)\}. \end{aligned}$$

Note 1. A distributed homogeneous track topology aggregation protocol $\mathcal{P}$ with domain $\mathcal{D}$ is correct for track topologies with a single node in the sink track if and only if

$$\mathscr{A}(\tau) = \mathcal{A}^k\left(V_f(n_1), \ldots, V_f(n_k)\right) \tag{6}$$

where τ is the message emitted by the sink node.

Also note that we implicitly assume a computation sequence (or *schedule*) that ensures data availability for all nodes. That is, whenever a node starts its computations, all the input data is available. Thus, we assume that all nodes connected to a node n are scheduled to transmit their data before n.

3 Verification of Track Topology Aggregation Protocols

To be able to provide a parameterized proof of correctness as defined in the previous section, we require the aggregation function to be an associative operator over the aggregation domain, so that there is a correspondence between the final aggregated values and the partial values transmitted by intermediate nodes. We also require the operator to provide a neutral element to be able to aggregate sensors whose communication fails or do not have data available in a uniform fashion.

Definition 5. *A family of aggregation functions $\mathcal{A}^k : \mathcal{D}_\circ^k \to \mathcal{D}$, $k \in \mathbb{N}_0$, over domain $\mathcal{D}$ is called* monoidal *with respect to $(\mathcal{D}_\circ, \oplus, \circ)$, if and only if $(\mathcal{D}_\circ, \oplus, \circ)$ is a monoid and*

$$\forall\, v_1, \ldots, v_k \in \mathcal{D}_\circ \bullet \mathcal{A}^k(v_1, \ldots, v_k) = v_1 \oplus \cdots \oplus v_k. \tag{7}$$

Bringing together the use of correction links and the transmission of correction requests, we introduce the concept of responsibility among nodes, defined as the confluence of both the availability of another node's data and of a correction request for that node.

Definition 6. *Let $T = (N, \longrightarrow, \rightsquigarrow, V)$ be a track topology and $f \in F_T$ a communication function.*

A node $n' \in N$ is called responsible *for node $n \in N$, denoted by $n \longrightarrow_g n'$, if and only if n' is an aggregator of n, communication from n to n' was successful according to f, and communication from n to the predecessors of n' in the correction relation was not successful according to f, i.e. if*

$$n \longrightarrow_g n' :\Longleftrightarrow n \longrightarrow_f n' \wedge \forall\, (n'', n) \in \rightsquigarrow^+ \bullet f(n, n'') = 0 \tag{8}$$

We say a node n has a working aggregation path *to node $n' \in N$ if and only if there is a sequence of responsible nodes between them, i.e., if $n \longrightarrow_g^* n''$.*

Note that working aggregation paths are unique (if existent). Protocol correctness can then be reformulated as the property that every node correctly aggregates the data of those nodes with a working aggregation path to it.

Lemma 2. *Let $\mathcal{A}$ be a monoidal aggregation function family with respect to $(\mathcal{D}_\circ, \oplus, \circ)$ and $\mathcal{P}$ a distributed homogeneous track topology aggregation protocol on track topologies $\mathcal{T}$ with domain $\mathcal{D}$.*

If the message of a node $n \in N$ encodes the $\oplus$-aggregation of the values of all nodes n' for which n is responsible, i.e. if

$$\forall\, T \in \mathcal{T}, f \in F_T \;\forall\, i \in ID_k \bullet \mathscr{A}(\tau_i) = \bigoplus_{n \longrightarrow_g^* n_i} V(n), \tag{9}$$

then $\mathcal{P}$ is correct wrt. $\mathcal{A}$.

Proof. Let $T \in \mathcal{T}$ be a topology of size k and $f \in F_T$ a communication function.

Let $n \in N$ be a node with a working aggregation path to a node $n' \in N_0$ in the sink layer, i.e. if $n \longrightarrow_g^* n'$, By Definition 6, there is a path from n to n' with successful communications, i.e. $n \longrightarrow_f^* n'$. Thus, by Definition 2, $V_f(n) = V(n)$.

Let $n \in N$ be a node with no working aggregation path to any node $n' \in N_0$ in the sink layer. Because such a path consists of primary and backup parents which are, by Definition 1, connected by the correction relation $\rightsquigarrow$. Thus there would exist a unique *first* parent for each track, and therefore n would have a working aggregation path, thus the value of n is not available at the sink track. Hence, by Definition 2, $V_f(n) = \circ$.

By Definition 4, the premise (9), and the premise that $\mathcal{A}$ is monoidal we obtain

$$\mathcal{P}(T,f) = \bigoplus_{n_i \in N_0} \bigoplus_{n \longrightarrow_g^* n_i} V(n) = \bigoplus_{n \in N} V_f(n) = \mathcal{A}^k(V_f(n_1), \ldots, V_f(n_k)) \tag{10}$$

thus $\mathcal{P}$ is correct. □

Now that we stated a property local to every node in a network, we introduce an abbreviation in the style of Hoare logic to denote the satisfaction of a postcondition given some precondition. We can in this form state the verification condition that must be satisfied in order to show protocol correctness. We choose this notation given that we intend to allow the verification of the implementation of the local aggregation functions against this property using software model checking.

Definition 7. *Let $\mathcal{P}$ be a distributed homogeneous track topology aggregation protocol and let P^k, Q^k be families of formulae over id, v, M_{id}, and τ_{id}.*
We write

$$\{P\}\ aggr\ \{Q\}$$

if and only if for for each $k \in \mathbb{N}_0$ and each valuation of id, v, M_{id} which satisfies P^k, that valuation extended by $\tau_{id} = aggr^k(id, v, M_{id})$ satisfies Q^k.

Having provided all the prerequisites, we can now state the main theorem of this work, where we reduce protocol correctness to the verification of a local node property. Assuming that all nodes in the network execute the same implementation of the local aggregation function *aggr*, we claim that if the local computation at a node ensures consistent aggregation up to that node, including the transmission of consistent correction requests, then the protocol is correct. We prove the soundness of this claim inductively.

Theorem 1. *Let $\mathcal{P}$ be a distributed homogeneous track topology aggregation protocol for the set $\mathcal{T}$ of track topologies with domain $\mathcal{D}$.*
If $\{P\}\ aggr\ \{Q\}$ with

$$P^k \equiv id \in ID_k \wedge v = V(n_{id})$$

and

$$\begin{aligned} Q^k \equiv \mathscr{A}(\tau_{id}) = V(n_{id}) \oplus & \bigoplus_{\substack{\tau_i \in M_{id} \\ \wedge((n_i, n_{id}) \in PA \vee (i \in \mathscr{E}(M_{id}) \wedge (n_i, n_{id}) \in SA))}} \mathscr{A}(\tau_i) \\ \wedge\, \mathscr{E}(\tau_{id}) = & \{i \mid \tau_i \notin M_{id} \wedge (n_i, n_{id}) \in PA\} \\ & \cup \{i \mid i \in \mathscr{E}(M_{id}) \wedge (\tau_i \notin M_{id} \vee (n_i, n_{id}) \notin \longrightarrow)\} \end{aligned}$$

then $\mathcal{P}$ is correct.

Proof (sketch). We show by double induction on the depth d of topologies and the size ℓ of their sink track that the set of messages emitted by the nodes according to the protocol satisfy the following properties:

– (■): For each node n, the emitted message comprises the aggregation of all nodes which n is (transitively) responsible for.
– (⧫): Correction for node n_j is initially requested by a node n_i if and only if n_i is the primary parent and there was no successful communication between the two.
– (★): A correction request for node n_j emitted by n_i implies that no predecessor of n_i in the correction relation was able to provide a correction.

Formally, we show

$$\begin{aligned}
&\forall\, d \in \mathbb{N}_0, \ell \in \mathbb{N}_0\ \forall\, T = (N, \longrightarrow, \rightsquigarrow, V) \in \mathcal{T} \bullet \\
&\quad \mathit{depth}(T) = d \wedge |N_0| = \ell \implies \\
&\qquad \forall\, n_i \in N \bullet \mathscr{A}(\tau_i) = \bigoplus_{n \longrightarrow_g^* n_i} V(n) && (■) \\
&\qquad\quad \wedge\, \forall\, j \bullet j \in \mathscr{E}(\tau_i) \setminus \mathscr{E}(M_i) \iff (n_j, n_i) \in \mathit{PA} \wedge f(n_j, n_i) = 0 && (⧫) \\
&\qquad\quad \wedge\, \forall\, j \in \mathscr{E}(\tau_i) \implies \forall\, (n', n_i) \in \rightsquigarrow^* \bullet n_j \longrightarrow n' \implies f(n_j, n') = 0. && (★)
\end{aligned}$$

Then (■) in particular implies (9), thus $\mathcal{P}$ is correct by Lemma 2. □

Discussion. The theorem presented allows us to eliminate most sources of infiniteness from the parameterized correctness proof for the studied protocols. Taking advantage of the fact that nodes have only local knowledge of the state of the system and the topology, we are able to implicitly profit from the resulting local symmetry of irregular networks.

The resulting verification problem is one where the system parameter k remains as a variable, and can thus be treated symbolically by verification tools. Alternatively, even a bounded correctness proof (up to some maximum size) can be provided by explicit model checking, provided that the data types used in the implementation are finite.

Developers of aggregation protocols can profit from the inductive proof provided. They can concentrate on verifying their particular implementations against our verification condition; a process that is less involved than providing a complete proof.

Some implicit assumptions were taken into account. We considered a single aggregation round, where nodes transmit only once. We thus assume that nodes are *memoryless* between aggregation rounds. This has, nonetheless, an advantage: systems are allowed to dynamically change their topology between aggregation rounds.

4 Case Study

We analyze the ridesharing protocol [9,10], proposed for use in DARPA's satellite cluster system F6 [3]. Nodes are arranged in tracks and reliability in the presence of failing communication links is improved by coordinating corrective

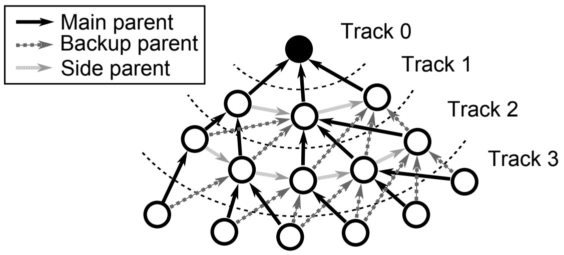


Fig. 2. Ridesharing topology

actions between nodes. We concentrate on a variant of the ridesharing protocol with reliable correction links. An initial attempt of the verification of the ridesharing protocol without considering the aggregation function, but the participation vector is presented in [8], for self-containment, we recall the protocol description here.

The goal of the protocol is to aggregate the readings of sensor arranged in a track graph towards a single, specially designated node called *sink*. Except from the sink, every node has one assigned parent (the *main* parent), which is responsible for aggregating its data. Additionally, other nodes in communication range are designated as fail-over aggregators (the *backup* parents) in case the communication between a node and its main parent fails.

The possible data paths from sensors to the sink form a directed, acyclic graph (DAG) with multiple paths through the track graph (See Fig. 2). Primary and backup edges are between adjacent tracks. There are also side edges within the same track. A side edge points to a node from its *side parent*. The main edges form a spanning tree with the sink as root.

Main parents transmit correction requests if the communication to their children fails. Those requests are relayed by the track nodes towards the backup parents connected to them using the reliable side links. If a backup parent receives a correction request for a backup child, and the data is available, it aggregates the data and stops propagating the error correction request.

We show the aggregation algorithm as proposed in [10] (see Algorithm 1). Each node in a given ridesharing topology has a unique identity. Input *id* gives the identity of the node and PC, BC, SP are the sets of primary children, backup children, and side parents of the node respectively. Input v gives the current sensor reading and *rcv* the messages received by *id*. The algorithm computes the message to be sent by *id*, given v and *rcv*.

Nodes transmit triples $\langle A, P, E\rangle$ with the accumulated sensor value A and two control boolean vectors of length $|N|$. The *participation vector* P indicates for each node whether its value is included in A and the *error vector* E indicates at each position, whether correction is required for the node at that position.

Aggregation starts by initializing A with zero and P and E with all zeroes. If node *id* has sensor data, it is aggregated to A and P is updated accordingly. We use $rcv[SP]$ to denote the bit-wise disjunction of the error vectors received

Algorithm 1. The *ridesharing* aggregation algorithm. [10]

```
input: $id$, $PC$, $BC$, $SP$, $v$, $rcv$
$A := 0$; $P := \bar{0}$; $E := \bar{0}$;
if $v \neq NULL$ then { $A := A + v$; $P[id] := 1$ } ;        // Aggregate local sensor reading
$E := rcv[SP]$;
foreach $c \in PC \cup BC$ do
    if $rcv[c] \neq undefined$ then
        if $c \in PA \vee (c \in SA \wedge E[c] = 1)$ then        // Aggregate received values
            $(A_c, P_c) := rcv[c]$; $A := A + A_c$; $P := P \mid P_c$; $E[c] := 0$;
        end
    else if $c \in PC$ then                                     // Request error correction
        $E[c] := 1$;
    end
end
return $(A, P, E)$;
```

from *id*'s side parents. Then, E comprises all requests for corrections. In the loop, the received messages from *id*'s children are processed: if *id* received the message from c and if c is a primary child or a backup child with a pending request for correction in E then c's data is aggregated, i.e. A is updated and the P vector becomes the disjunction of the incoming P vectors. If *id* did not receive the message from primary child c, it flags a request for the correction of c and leaves A and P unchanged.

4.1 Verification of the Ridesharing Protocol

We show how the ridesharing protocol fits into the framework presented in Section 2. First, we show that the topologies used are track topologies.

- The set of nodes $N = \{n_1, \ldots, n_k\}$ is the set of nodes present in a ridesharing topology, there is a single sink node in the sink track $N_0 = \{n_k\}$,
- The main parent relation is the primary aggregation relation PA. It connects nodes between adjacent tracks and forms a tree with n_k as root.
- The backup parent relation is the secondary aggregation relation SA. It forms an acyclic directed graph (N, SA).
- The side parent relation is the correction relation $\rightsquigarrow$. It connects nodes sequentially.
- The *track* function can be defined with respect to PA since it forms a spanning tree:
$$track(n) = \begin{cases} 0 & \text{iff } \nexists\,(n, n') \in PA \\ track(n') + 1 & \text{iff } \exists\,(n, n') \in PA \end{cases}$$
- each node n_i has access to its sensor value v_i from the domain $\mathcal{D}$ of bounded integers, if the sensor does not have a value, $v_i = 0$ is used. Thus the labeling function is defined as: $V = \{n_i \mapsto v_i \mid 1 \leq i \leq k\}$.

Second, we show that the protocol is distributed homogeneous.

- The domain is the type of the sensor readings, we assume it to be bounded integer numbers: $\mathcal{D} = \{-2^n, \ldots, 2^n - 1\}$ for some positive integer n.

- The aggregation operator is the sum operator. I.e., $\oplus = +$.
- The domain, aggregation operator and neutral element $(\mathcal{D}, +, 0)$ form a monoid. Thus the aggregation function $\mathcal{A}^k(v_1, \ldots, v_k) = v_1 + \cdots + v_k$ is monoidal.
- The set of messages is the set of transmissions of the nodes $\tau_i = \langle A_i, P_i, E_i \rangle$ where A_i, P_i, E_i are the values calculated by *ridesharing* at node i. Thus, $Msg_k = \mathcal{D} \times \mathbb{B}^k \times \mathbb{B}^k$.
- The value decoder function reads the aggregation value directly from a message, i.e., $\mathscr{A}(\tau_i) = A_i$. The correction requests can also be read directly from the transmitted messages: $\mathscr{E}(\tau_i) = \{j \in ID_k \mid E_i[j] = 1\}$.
- The local aggregation function is Algorithm 1. I.e.,

$$aggr^k(i, v, M) = ridesharing(i, PA(i), SA(i), SP(i), v, M)$$

  where $PA(i) = \{j \in ID_k \mid (j, i) \in PA\}$. $SA(i)$ and $SP(i)$ are defined analogously w.r.t. SA and $\rightsquigarrow$. M is defined with respect to a communication function as in Def. 4.

We can thus apply the verification theorem to the ridesharing protocol. It is left to verify whether the formula

$$\{P\}\ ridesharing(i, PA(i), SA(i), \rightsquigarrow (i), v, M)\ \{Q\}$$

is valid for P and Q as defined in Theorem 1.

To that end, we developed a model including our formalization from Sec. 2 using Boogie [2]. The topology and communication functions are represented by axioms. Additional axioms represent the properties of the system analyzed. The aggregation algorithm is input directly in imperative form by representing boolean arrays as functions of type $array : N \rightarrow \mathbb{B}$. Instead of bounded integers, we used unbounded integers to simplify the model, thus introducing the assumption that the actual data types used in an implementation are enough to accommodate the aggregation values without overflowing. Simple invariants for the loop in the algorithm were necessary: framing conditions for the loop variables and the fact that consistency is preserved across iterations of the loop was sufficient. P and Q were expressed in terms of the protocol variables and used as pre- and postconditions of the algorithm respectively[1].

The verification took 1.05 seconds to verify 35 partial verification conditions (including sanity checks), while using approximately 13 MB of memory. Boogie was able to show that the verifications conditions are all valid. We thus have shown correctness of the ridesharing protocol with reliable side links.

Discussion. Due to the symbolic representation of the parametric set of nodes, the theorem prover used by Boogie requires only a finite number of cases to show the validity of the generated verification conditions fully automatically.

recall all assumptions

[1] Model available at `http://www.informatik.uni-freiburg.de/~arenis/forte13/`

Thanks to the result that the ridesharing protocol is correct, in the sense of Definition 3, a reliability measure can be simply computed from the failure probability of the inter-track radio links as follows:

Let $p(n_i, n_j)$ be the probability of node n_j successfully receiving the transmission of a connected node n_i. The probability of the data of some node n being successfully aggregated is then equal to the probability of all links in each possible aggregation path failing:

$$1 - \prod_{\pi \in paths(n)} \left(1 - \prod_{1 \le j < |\pi|} p\left(\pi[j], \pi[j+1]\right) \right)$$

where $paths(n)$ are the sequences $\pi = n_1, \ldots, n_m$ of nodes starting at node n, connected by the aggregation relation and ending in the sink node. The length $|\pi| = m$. The expression $\pi[i]$ refers to the i-th node on the path π.

If all links have the same success probability p, then the probability of a node n in track t successfully being aggregated at the sink simplifies to

$$1 - \left(1 - p^{track(n)} \right)^{|paths(n)|}$$

assuming that link failures are independent.

These results thus reduce the need to use simulation tools to determine the overall reliability of the ridesharing protocol.

5 Related Work

The parameterized analysis of safety properties is undecidable in the general case [1]. The problem of verifying properties for parameterized systems has been approached by several authors. We discuss a few that we feel are the closest to our line of work.

We can best situate our work in the line of work of Namjoshi et al. [14] for the verification of irregular networks. Although we concentrate on asynchronous systems, the verification condition presented in this work can be viewed as a particular case of an inductive, compositional invariant, though used in the context of irregular networks with dynamic links. We ensure the local symmetry of aggregation networks with redundancy by explicitly considering three types of edges and restricting the analysis to systems where inter-node communication occurs after computation, without interleaving. We enrich the computation model by explicitly including the network dynamic as part of the system state.

The particular problem of network dynamic in the form of link and node failures has been the subject of recent efforts [5]. We present an instance of verification of broadcast networks with link unreliability, which can simulate node failures, for the special case of aggregation protocols.

The method of “network grammars” [4], where regular families of networks are described using context-free grammars is used to verify parameterized systems by using abstraction. Several model checking problems are derived, which, if solved, deliver a proof of correctness. We do not explicitly perform any kind of abstraction on the states of the local aggregation programs, leaving that possibility open for the method used to establish the satisfaction of our verification condition.

Similar methods rely on finding so called network invariants [11,15], overapproximations of the sets of reachable states of the systems, and proving that they imply the property being verified. The focus lies on automatically finding those invariants through model checking. We distance ourselves from the automatic finding of invariants and perform explicit induction using a fixed invariant, delivering a proof rule for the class of aggregation protocols with irregular topology. Explicit induction has been also used for the verification of regular networks [6].

The systems we consider in this work are asynchronous systems where computation steps of network components are considered atomic. The systems could be, however, considered as an instance of bounded-data parameterized systems in the sense of [15].

This work is a follow-up of the initial analysis of the protocol from our case study [8]. We improve the analysis by extending it to the use of arbitrary duplicate-sensitive aggregation functions, eliminating the need for the explicit transmission of a vector of aggregated nodes. Here, we provide a complete, formal inductive proof and, as a result, simplify the resulting verification conditions. The software model checking step of the verification is thus simplified, improving the chances of obtaining a verification problem within the capabilities of current software verification tools.

6 Conclusions

We have presented an inductive, parameterized proof of correctness for track topology aggregation protocols. We reduce the problem to the verification of a single verification consistency condition on the nodes' aggregation algorithm, which can be automatically checked using state of the art software verification tools.

We identify a class of aggregation protocols that allows us to illustrate the feasibility of reducing the verification of parameterized, asynchronous systems with irregular topologies and dynamic link (and thus node) reliability to establishing local consistency properties.

Possible directions for future work are the relaxation of the constraints for the systems verified –to enlarge the class of systems covered–, such as the reliability of the correction infrastructure and the acyclicity of some of the node connection relations.

Additionally, the use of our verification conditions as a template for the automatic generation of inductive node invariants can be explored, such that given decoding functions for a protocol messages, the verification task is further automated.

Finally, criteria for the decidability of the resulting verification problem can be studied. This would lead to the identification of a decidable class of parameterized systems.

References

1. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. 22(6), 307–309 (1986)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Brown, O., Eremenko, P.: The value proposition for Fractionated space architectures. In: AIAA Space 2006. No. 7506. AIAA (2006)
4. Clarke, E.M., Grumberg, O., Jha, S.: Verifying parameterized networks. ACM Trans. Program. Lang. Syst. 19(5), 726–750 (1997)
5. Delzanno, G., Sangnier, A., Zavattaro, G.: Verification of ad hoc networks with node and communication failures. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 235–250. Springer, Heidelberg (2012)
6. Emerson, E.A., Namjoshi, K.S.: On reasoning about rings. Int. J. Found. Comput. Sci. 14(4), 527–550 (2003)
7. Feng, J., Eager, D.L., Makaroff, D.J.: Aggregation protocols for high rate, low delay data collection in sensor networks. In: Fratta, L., Schulzrinne, H., Takahashi, Y., Spaniol, O. (eds.) NETWORKING 2009. LNCS, vol. 5550, pp. 26–39. Springer, Heidelberg (2009)
8. Feo-Arenis, S., Westphal, B.: Formal verification of a parameterized data aggregation protocol. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 428–434. Springer, Heidelberg (2013)
9. Gobriel, S., Khattab, S., Mossé, D., Brustoloni, J., Melhem, R.: Ridesharing: Fault tolerant aggregation in sensor networks using corrective actions. In: IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, pp. 595–604 (2006)
10. Iskander, M.K., Lee, A.J., et al.: Privacy and robustness for data aggregation in wireless sensor networks. In: 17th ACM Conference on Computer and Communications Security, pp. 699–701. ACM (2010)
11. Kesten, Y., Pnueli, A., Shahar, E., Zuck, L.D.: Network invariants in action. In: Brim, L., Jančar, P., Křetínský, M., Kučera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 101–115. Springer, Heidelberg (2002)
12. Liu, M., Gong, H.G., Mao, Y.C., Chen, L.J., Xie, L.: A distributed energy-efficient data gathering and aggregation protocol for wireless sensor networks. Journal of Software 16(12), 2106–2116 (2005)
13. Montresor, A., Jelasity, M., Babaoglu, O.: Robust aggregation protocols for large-scale overlay networks. In: 2004 International Conference on Dependable Systems and Networks, pp. 19–28. IEEE (2004)
14. Namjoshi, K.S., Trefler, R.J.: Uncovering symmetries in irregular process networks. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 496–514. Springer, Heidelberg (2013)
15. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001)

Monitoring Networks through Multiparty Session Types*

Laura Bocchi[1], Tzu-Chun Chen[2], Romain Demangeon[2], Kohei Honda[2], and Nobuko Yoshida[3]

[1] University of Leicester
[2] Queen Mary, University of London
[3] Imperial College London

Abstract. In large-scale distributed infrastructures, applications are realised through communications among distributed components. The need for methods for assuring safe interactions in such environments is recognized, however the existing frameworks, relying on centralised verification or restricted specification methods, have limited applicability. This paper proposes a new theory of *monitored* π-calculus with dynamic usage of *multiparty session types* (MPST), offering a rigorous foundation for safety assurance of distributed components which asynchronously communicate through multiparty sessions. Our theory establishes a framework for semantically precise decentralised run-time enforcement and provides reasoning principles over monitored distributed applications, which complement existing static analysis techniques. We introduce asynchrony through the means of explicit routers and global queues, and propose novel equivalences between networks, that capture the notion of interface equivalence, i.e. equating networks offering the same services to a user. We illustrate our static-dynamic analysis system with an ATM protocol as a running example and justify our theory with results: satisfaction equivalence, local/global safety and transparency, and session fidelity.

1 Introduction

One of the main engineering challenges for distributed systems is the comprehensive verification of distributed software without relying on ad-hoc and expensive testing techniques. Multiparty session types (MPST) is a typing discipline for communication programming, originally developed in the π-calculus [15,1,3,11,12,7] towards tackling this challenge. The idea is that applications are built starting from units of design called sessions. Each type of session, involving multiple roles, is first modelled from a global perspective (*global type*) and then projected onto *local types*, one for each role involved. As a verification method, the existing MPST systems focus on static type checking of endpoint processes against local types. The standard

* This work has been partially sponsored by the project Leverhulme Trust Award Tracing Networks, Ocean Observatories Initiative and EPSRC EP/K011715/1, EP/G015635/1 and EP/G015481/1.

properties enjoyed by well-typed processes are communication safety (all processes conform to globally agreed communication protocols) and freedom from deadlocks.

The direct application of the theoretical MPST techniques to the current practice, however, presents a few obstacles. Firstly, the existing type systems are targeted at calculi with first class primitives for linear communication channels and communication-oriented control flow; the majority of mainstream engineering languages would need to be extended in this sense to be suitable for syntactic session type checking. Unfortunately, it is not always straightforward to add these features to the specific host languages (e.g. linear resource typing for a very liberal language like C). Furthermore, the executable processes in a distributed system may be implemented in different languages. Secondly, for domains where dynamically typed or untyped languages are popular (e.g., Web programming), or in multi-organizational scenarios, the introduction of static typing infrastructure to support MPST may not be realistic.

This paper proposes a theoretical system addressing the above issues by enabling both static *and* dynamic verification of communicating processes. The aim is to capture the decentralised nature of distributed application development, providing better support for heterogeneous distributed systems by allowing components to be independently implemented, using different languages, libraries and programming techniques, as well as being independently verified, either statically or dynamically, while retaining the strong global safety properties of statically verified homogeneous systems.

This work is motivated in part by our ongoing collaboration with the Ocean Observatories Initiative (OOI) [17], a project to establish cyberinfrastructure for the delivery, management and analysis of scientific data from a large network of ocean sensor systems. Their architecture relies on the combination of high-level protocol specifications (to express how the infrastructure services should be used) and distributed run-time monitoring to regulate the behaviour of third-party applications in the system.

A Formal Theory for Static/Dynamic Verification. Our framework is based on the idea that, if each endpoint is *independently* verified (statically or dynamically) to conform to their local protocols, then the global protocol is respected *as a whole*. To this goal, we propose a new formal model and bisimulation theories of heterogeneous networks of monitored and unmonitored processes.

For the first time, we make explicit the *routing mechanism* implicitly present inside the MPST framework: in a session, messages are sent to abstract roles (e.g. to a Seller) and the router, a dynamically updated component of the network, translates these roles into actual addresses.

By taking this feature into account when designing novel equivalences, our formal model can relate networks built in different ways (through different distributions or relocations of services) but offering the same *interface* to an external observer. The router, being in charge of associating roles with principals, hides to an external user the internal composition of a network: what distinguishes two networks is not their structure but the services they are able to perform, or more precisely, the local types they offer to the outside.

We formally define a satisfaction relation to express when the behaviour of a network conforms to a global specification and we prove a number of properties of our model. *Local safety* states that a monitored process respects its local protocol, i.e. that dynamic verification by monitoring is sound, while *local transparency* states that a monitored process has equivalent behaviour to an unmonitored but well-behaved process, e.g. statically verified against the same local protocol. *Global safety* states that a system satisfies the global protocol, provided that each participant behaves as if monitored, while *global transparency* states that a fully monitored network has equivalent behaviour to an unmonitored but well-behaved network, i.e. in which all local processes are well-behaved against the same local protocols. *Session fidelity* states that, as all message flows of a network satisfy global specifications, whenever the network changes because some local processes take actions, all message flows continue to satisfy global specifications. Together, these properties justify our framework for decentralised verification by allowing monitored and unmonitored processes to be safely mixed while preserving protocol conformance for the entire network. Technically, these properties also ensure the coherence of our theory, by relating the satisfaction relations with the semantics and static validation procedures.

Paper Summary and Contributions. § 2 introduces the formalisms for protocol specifications (§ 2.1) and networks (§ 2.2) used to provide a formal framework for monitored networks based on π-calculus processes and protocol-based runtime enforcement through monitors. § 3 introduces: a semantics for specifications (§ 3.1), a novel behavioural theory for compositional reasoning over monitored networks through the use of equivalences (bisimilarity and barbed congruence) and the satisfaction relation (§ 3.2). § 3.4 establishes key properties of monitored networks, namely local/global safety, transparency, and session fidelity. We discuss future and related work in § 4. The proofs can be found in [2].

2 Types, Processes and Networks: A Formal Presentation

This section and the next one provide a theoretical basis for protocol-centred safety assurance. We first summarise the syntax of MPSTs (multiparty session types) annotated with logical assertions [3]. We then introduce a novel *monitored session calculus* as a variant of the π-calculus, modelling distributed dynamic components (whose behaviours are realised by processes) and monitors, all residing in global networks.

2.1 Multiparty Session Types with Assertions

Multiparty session types with assertions [3] are abstract descriptions of the structure of interactions among the participants of a multiparty session, specifying potential flows of messages, the conditions under which these interactions may be done, and the constraints on the communicated values. In this framework, *global types with assertions*, or just *global types*, describe multiparty sessions from a network perspective. From global types one can derive, through *endpoint projection*, *local types with assertions*, or just *local types*, describing the protocol from the perspective of a single endpoint.

$$
\begin{array}{l}
A ::= \mathsf{tt} \;\big|\; \mathsf{ff} \;\big|\; e_1 = e_2 \;\big|\; e_1 < e_2 \;\big|\; \neg A \;\big|\; A_1 \wedge A_2 \;\big|\; A_1 \vee A_2 \\
e \;::= v \;\big|\; e_1 + e_2 \;\big|\; e_1 - e_2 \;\big|\; e_1 * e_2 \;\big|\; e_1 \;\mathtt{mod}\; e_2 \qquad S ::= \mathsf{bool} \;\big|\; \mathsf{int} \;\big|\; \mathsf{string} \\
G ::= \mathtt{r}_1 \to \mathtt{r}_2 : \{l_i(x_i : S_i)\{A_i\}.G_i\}_{i\in I} \;\big|\; G_1 \mid G_2 \;\big|\; G_1 ; G_2 \;\big|\; \mu \mathsf{t}.G \;\big|\; \mathsf{t} \;\big|\; \epsilon \;\big|\; \mathsf{end} \\
T ::= \mathtt{r}!\{l_i(x_i : S_i)\{A_i\}.T_i\}_{i\in I} \;\big|\; \mathtt{r}?\{l_i(x_i : S_i)\{A_i\}.T_i\}_{i\in I} \;\big|\; T_1 \mid T_2 \;\big|\; T_1 ; T_2 \;\big|\; \\
\qquad \mu \mathsf{t}.T \;\big|\; \mathsf{t} \;\big|\; \epsilon \;\big|\; \mathsf{end}
\end{array}
$$

The syntax of the global types $(G, G', \ldots)$ and local types $(T, T', \ldots)$ is given above. The grammar is based on [3,12] extended with parallel threads, which also require sequential composition to merge parallel threads as in [19]. We let values $v, v', \ldots$ range over boolean constants, numerals and strings, and $e, e', \ldots$ range over first-order expressions. For expressing constraints, we use logical predicates, or *assertions*, ranged over by $A, A', \ldots$, following the grammar given above, although other decidable logics could be used.[1] The *sorts* of exchanged values $(S, S', \ldots)$ consists of atomic types.

Global Types with Assertions. $\mathtt{r}_1 \to \mathtt{r}_2 : \{l_i(x_i : S_i)\{A_i\}.G_i\}_{i\in I}$ models an interaction where role $\mathtt{r}_1$ sends role $\mathtt{r}_2$ one of the *branch labels* l_i, as well as a value denoted by an *interaction variable* x_i of sort S_i. Interaction variable x_i binds its occurrences in A_i and G_i. A_i is the assertion which needs to hold for $\mathtt{r}_1$ to select l_i, and which may constrain the values instantiating x_i. $G_1 \mid G_2$ specifies two parallel sessions, and $G_1; G_2$ denotes sequential composition (assuming that G_1 does not include end). μt.G is a recursive type, where t is guarded in G in the standard way, ϵ is the inaction for absence of communication, and end ends the session.

*Example 1 (*ATM*: the global type).* We present global type $G_{\mathtt{ATM}}$ that specifies an *ATM* scenario. Each session of *ATM* involves three roles: a client (C), the payment server (S) and a separate authenticator (A).

$$
\begin{array}{rl}
G_{\mathtt{ATM}} = & \mathtt{C} \to \mathtt{A} : \{\, \mathtt{Login}(x_i : \mathsf{string})\{\mathsf{tt}\}. \\
& \mathtt{A} \to \mathtt{S} : \{\, \mathtt{LoginOK}()\{\mathsf{tt}\}.\ \mathtt{A} \to \mathtt{C} : \{\mathtt{LoginOK}()\{\mathsf{tt}\}.\ G_{\mathsf{Loop}}\}, \\
& \qquad\qquad \mathtt{LoginFail}()\{\mathsf{tt}\}.\ \mathtt{A} \to \mathtt{C} : \{\mathtt{LoginFail}()\{\mathsf{tt}\}.\ \mathsf{end}\}\}\} \\
G_{\mathsf{Loop}} = & \mu\ \mathtt{LOOP}. \\
& \mathtt{S} \to \mathtt{C} : \{\, \mathtt{Account}(x_b : \mathsf{int})\{x_b \geq 0\}. \\
& \mathtt{C} \to \mathtt{S} : \{\, \mathtt{Withdraw}(x_p : \mathsf{int})\{x_p > 0 \wedge x_b - x_p \geq 0\}.\ \mathtt{LOOP}, \\
& \qquad\qquad \mathtt{Deposit}(x_d : \mathsf{int})\{x_d > 0\}.\ \mathtt{LOOP}, \\
& \qquad\qquad \mathtt{Quit}()\{\mathsf{tt}\}.\mathsf{end}\}\}
\end{array}
$$

At the start of the session C sends its login details x_i to A, then A informs S and C whether the authentication is successful, by choosing either the branch

[1] We use a logic without quantifiers, contrary to [3], to simplify the presentation and because *monitorability*, defined later in this section, makes them unnecessary.

with label `LoginOK` or `LoginFail`. In the former case `C` and `S` enter a transaction loop specified by G_{Loop}. In each iteration `S` sends `C` the amount x_b available in the account, which must be non negative. Next, `C` has three choices: `Withdraw` withdraws an amount x_p (x_p must be positive and not exceed the current amount x_b) and repeats the loop, `Deposit` deposits a positive amount x_d in the account and repeats the loop, and `Quit` ends the session.

We consider global types that satisfy the consistency conditions defined in [11,3,12] which rule out, for instance, protocols where interactions have causal relations that cannot be enforced (e.g., we write $\mathtt{r}_A \to \mathtt{r}_B : \mathtt{l}_1()\{\mathsf{tt}\} \mid \mathtt{r}_C \to \mathtt{r}_D : \mathtt{l}_2()\{\mathsf{tt}\}$ instead of $\mathtt{r}_A \to \mathtt{r}_B : \mathtt{l}_1()\{\mathsf{tt}\}.\mathtt{r}_C \to \mathtt{r}_D : \mathtt{l}_2()\{\mathsf{tt}\}$). In addition we assume *monitorability* requiring that in all the interactions of the form $\mathtt{r} \to \mathtt{r}' : \mathtt{l}(x : \mathsf{S})\{A\}$ occurring in a global type G both $\mathtt{r}$ and $\mathtt{r}'$ *know* (i.e., have sent or received in a previous or in this interaction) the free variables in A.

Local Types with Assertions. Each local type T is associated with a role taking part in a session. Local type $\mathtt{r}!\{l_i(x_i : S_i)\{A_i\}.T_i\}_{i\in I}$ models an interaction where the role under consideration sends $\mathtt{r}$ a branch label l_i and a message denoted by an interaction variable x_i of sort S_i. Its dual is the receive interaction $\mathtt{r}?\{l_i(x_i : S_i)\{A_i\}.T_i\}_{i\in I}$. The other local types are similar to the global types.

One can derive a set of local types T_i from a global type G by *endpoint projection*, defined as in [3]. We write $G \restriction \mathtt{r}$ for the projection of G onto role $\mathtt{r}$. We illustrate the main projection rule, which is for projecting a global type modelling an interaction. Let G be $(\mathtt{r} \to \mathtt{r}' : \{l_i(x_i : S_i)\{A_i\}.G_i\}_{i\in I})$; the projection of G on $\mathtt{r}$ is $\mathtt{r}'!\{l_i(x_i : S_i)\{A_i\}.(G_i \restriction \mathtt{r})\}_{i\in I}$, and the projection of G on $\mathtt{r}'$ is $\mathtt{r}?\{l_i(x_i : S_i)\{A_i\}.(G_i \restriction \mathtt{r}')\}_{i\in I}$. The other rules are homomorphic, following the grammar of global types inductively.

*Example 2 (*ATM*: the local type of* `C`*).* We present the *local type* $T_{\mathtt{C}}$ obtained by projecting G_{ATM} on role `C`.

$$T_{\mathtt{C}} = \mathtt{A}!\{\mathtt{Login}(x_i : \mathsf{string})\{\mathsf{tt}\}.\ \mathtt{A}?\{\mathtt{LoginOK}()\{\mathsf{tt}\}.\ T_{\text{Loop}}\ \ \mathtt{LoginFail}()\{\mathsf{tt}\}.\ \mathsf{end}\}\}$$

$$T_{\text{Loop}} = \mu\,\mathtt{LOOP}.\ \mathtt{S}?\{\mathtt{Account}(x_b : \mathsf{int})\{x_b \geq 0\}.\ \mathtt{S}!\{\mathtt{Withdraw}(x_p : \mathsf{int})\{x_p > 0 \wedge x_b - x_p \geq 0\}.\ \mathtt{LOOP},\ \mathtt{Deposit}(x_d : \mathsf{int})\{x_d > 0\}.\mathtt{LOOP},\ \mathtt{Quit}()\{\mathsf{tt}\}.\mathsf{end}\}\}$$

$T_{\mathtt{C}}$ specifies the behaviour that `C` should follow to meet the contract of global type G_{ATM}. $T_{\mathtt{C}}$ states that `C` should first authenticate with `A`, then receive the `Account` message from `S`, and then has the choice of sending `Withdraw` (and enact the recursion), or `Deposit` (and enact the recursion) or `Quit` (and end the session).

2.2 Formal Framework of Processes and Networks

In our formal framework, each distributed application consists of one or more sessions among *principals.* A principal with behaviour P and name α is represented as $[P]_\alpha$. A *network* is a set of principals together with a (unique) *global* *transport*, which abstractly represents the communication functionality of a distributed system. The syntax of processes, principals and networks is given below, building on the multiparty session π-calculus from [1].

$$
\begin{array}{l}
P ::= \overline{a}\langle s[\mathtt{r}] : T\rangle \;\big|\; a(y[\mathtt{r}]{:}T).P \;\big|\; k[\mathtt{r}_1,\mathtt{r}_2]!l\langle e\rangle \;\big|\; k[\mathtt{r}_1,\mathtt{r}_2]?\{l_i(x_i).P_i\}_{i\in I} \;\big| \\
\qquad \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \;\big|\; P \mid Q \;\big|\; \mathbf{0} \;\big|\; \mu X.P \;\big|\; X \;\big|\; P;Q \;\big|\; (\nu a)\ P \;\big|\; (\nu s)P \\
N ::= [P]_\alpha \;\big|\; N_1 \mid N_2 \;\big|\; \mathbf{0} \;\big|\; (\nu a)N \;\big|\; (\nu s)N \;\big|\; \langle r\ ;\ h\rangle \\
r ::= a \mapsto \alpha \;\big|\; s[\mathtt{r}] \mapsto \alpha \qquad h ::= m\cdot h \;\big|\; \emptyset \qquad m ::= \overline{a}\langle s[\mathtt{r}] : T\rangle \;\big|\; s\langle \mathtt{r}_1,\mathtt{r}_2, l\langle v\rangle\rangle
\end{array}
$$

$\mathtt{r},\mathtt{r}_1,\ldots$	roles	$s, s',\ldots$	session names	$X, Y,\ldots$	process variables
$a, b,\ldots$	shared names	$x, y,\ldots$	variables	$P, Q,\ldots$	processes
$\alpha,\beta,\ldots$	principal names	$N, N',\ldots$	networks		

Processes. Processes are ranged over by $P, P',\ldots$ and communicate using two types of channel: *shared channels* (or shared names) used by processes for sending and receiving invitations to participate in sessions, and *session channels* (or session names) used for communication *within* established sessions. One may consider session names as e.g., URLs or service names.

The *session invitation* $\overline{a}\langle s[\mathtt{r}] : T\rangle$ invites, through a shared name a, another process to play $\mathtt{r}$ in a session s. The *session accept* $a(y[\mathtt{r}] : T).P$ receives a session invitation and, after instantiating y with the received session name, behaves in its continuation P as specified by local type T for role $\mathtt{r}$. The *selection* $k[\mathtt{r}_1,\mathtt{r}_2]!l\langle e\rangle$ sends, through session channel k (of an established session), and as a sender $\mathtt{r}_1$ and to a receiver $\mathtt{r}_2$, an expression e with label l. The *branching* $k[\mathtt{r}_1,\mathtt{r}_2]?\{l_i(x_i).P_i\}_{i\in I}$ is ready to receive one of the labels and a value, then behaves as P_i after instantiating x_i with the received value. We omit labels when I is a singleton. The *conditional*, *parallel* and *inaction* are standard. The *recursion* $\mu X.P$ defines X as P. Processes $(\nu a)P$ and $(\nu s)P$ hide shared names and session names, respectively.

Principals and Network. A *principal* $[P]_\alpha$, with its process P and name α, represents a unit of behaviour (hence verification) in a distributed system. A *network* N is a collection of principals with a unique global transport.

A *global transport* $\langle r\ ;\ h\rangle$ is a pair of a routing table which delivers messages to principals, and a global queue. Messages between two parties inside a single session are ordered (as in a TCP connection), otherwise unordered. More precisely, in $\langle r\ ;\ h\rangle$, h is a *global queue*, which is a sequence of *messages* $\overline{a}\langle s[\mathtt{r}] : T\rangle$ or $s\langle \mathtt{r}_1,\mathtt{r}_2, l\langle v\rangle\rangle$, ranged over by m. These m represent messages-in-transit, i.e. those messages which have been sent from some principals but have not yet been delivered. The *routing table* r is a finite map from session-roles and shared names to principals. If, for instance, $s[\mathtt{r}] \mapsto \alpha \in r$ then a message for $\mathtt{r}$ in session s will be delivered to principal α.

Let $n, n',\ldots$ range over shared and session channels. A network N which satisfies the following conditions is *well-formed*: (1) N contains at most one global transport; (2) two principals in N never have the same principal name; and (3) if $N \equiv (\nu\tilde{n})(\prod_i [P_i]_{\alpha_i} | \langle r \;;\; h\rangle)$, each free shared or session name in P_i and h occurs in $\tilde{n}$ (we use $\prod_i P_i$ to denote $P_1 \mid P_2 \cdots \mid P_n$).

Semantics. The reduction relation for dynamic networks is generated from the rules below, which model the interactions of principals with the global queue.

$$[\overline{a}\langle s[\mathtt{r}] : T\rangle]_\alpha \mid \langle r \;;\; h\rangle \longrightarrow [\mathbf{0}]_\alpha \mid \langle r \;;\; h \cdot \overline{a}\langle s[\mathtt{r}] : T\rangle\rangle \qquad \lfloor\text{REQ}\rfloor$$

$$[a(y[\mathtt{r}] : T).P]_\alpha \mid \langle r \;;\; \overline{a}\langle s[\mathtt{r}] : T\rangle \cdot h\rangle \longrightarrow [P[s/y]]_\alpha \mid \langle r{\cdot}s[\mathtt{r}] \mapsto \alpha \;;\; h\rangle\ ^{\dagger} \qquad \lfloor\text{ACC}\rfloor$$

$$[s[\mathtt{r}_1,\mathtt{r}_2]!l_j\langle v\rangle]_\alpha \mid \langle r \;;\; h\rangle \longrightarrow [\mathbf{0}]_\alpha \mid \langle r \;;\; h \cdot s\langle \mathtt{r}_1,\mathtt{r}_2,l_j\langle v\rangle\rangle\rangle\ ^{\dagger\dagger} \qquad \lfloor\text{SEL}\rfloor$$

$$[s[\mathtt{r}_1,\mathtt{r}_2]?\{l_i(x_i).P_i\}_i]_\alpha \mid \langle r \;;\; s\langle \mathtt{r}_1,\mathtt{r}_2,l_j\langle v\rangle\rangle \cdot h\rangle \longrightarrow [P_j[v/x_j]]_\alpha \mid \langle r \;;\; h\rangle\ ^{\dagger\dagger\dagger} \qquad \lfloor\text{BRA}\rfloor$$

$$[\mathsf{if\ tt\ then}\ P\ \mathsf{else}\ Q]_\alpha \longrightarrow [P]_\alpha \qquad [\mathsf{if\ ff\ then}\ P\ \mathsf{else}\ Q]_\alpha \longrightarrow [Q]_\alpha \qquad \lfloor\text{CND}\rfloor$$

$$\frac{[P]_\alpha \mid N \longrightarrow [P']_\alpha \mid N'}{[\mathcal{E}(P)]_\alpha \mid N \longrightarrow [\mathcal{E}(P')]_\alpha \mid N'} \qquad \frac{e \longrightarrow e'}{[\mathcal{E}(e)]_\alpha \longrightarrow [\mathcal{E}(e')]_\alpha} \qquad \frac{N \longrightarrow N'}{\mathcal{E}(N) \longrightarrow \mathcal{E}(N')} \qquad \lfloor\text{CTX}\rfloor$$

$$\dagger : r(a) = \alpha \qquad \dagger\dagger : r(s[\mathtt{r}_2]) \neq \alpha \qquad \dagger\,\dagger\dagger : r(s[\mathtt{r}_2]) = \alpha$$

$$\mathcal{E} ::= (\ ) \;\big|\; \mathcal{E} \mid P \;\big|\; (\nu s)\mathcal{E} \mid (\nu a)\mathcal{E} \mid \mathcal{E};P \mid \mathcal{E} \mid N \mid \text{if } \mathcal{E} \text{ then } P \text{ else } Q \mid s[\mathtt{r}_1,\mathtt{r}_2]!l\langle\mathcal{E}\rangle$$

Rule $\lfloor$REQ$\rfloor$ places an invitation in the global queue. Dually, in $\lfloor$ACC$\rfloor$, a process receives an invitation on a shared name from the global queue, assuming a message on a is to be routed to α. As a result, the routing table adds $s[\mathtt{r}] \mapsto \alpha$ in the entry for s. Rule $\lfloor$SEL$\rfloor$ puts in the queue a message sent from $\mathtt{r}_1$ to $\mathtt{r}_2$, which selects label l_j and carries v, if it is not going to be routed to α (i.e. sent to self). Dually, $\lfloor$BRA$\rfloor$ gets a message with label l_j from the global queue, so that the j-th process P_j receives value v. The reduction is also defined modulo the structural congruence $\equiv$ defined by the standard laws over processes/networks, the unfolding of recursion ($\mu X.P \equiv P[\mu X.P/X]$) and the associativity and commutativity and the rules of message permutation in the queue [15,11]. The other rules are standard.

*Example 3 (*ATM*: an implementation).* We now illustrate the processes implementing the client role of the *ATM* protocol. We let $P_{\mathtt{C}}$ be the process implementing $T_{\mathtt{C}}$ (from Example 2) and communicating on session channel s.

$$\begin{array}{l|l}
P_{\mathtt{C}} = s[\mathtt{C},\mathtt{A}]!\mathtt{Login}(\mathit{alice\_pwd}123); & P''_{\mathtt{C}} = \mathsf{if}\ \mathit{getmore}() \wedge (x_b \geq 10) \\
\qquad s[\mathtt{A},\mathtt{C}]?\{\mathtt{LoginOK}();\mu X.P'_{\mathtt{C}}, \mathtt{LoginFail}().\mathbf{0}\} & \qquad \mathsf{then}\ s[\mathtt{C},\mathtt{S}]!\mathtt{Withdraw}(10);X \\
P'_{\mathtt{C}} = s[\mathtt{S},\mathtt{C}]?\mathtt{Account}(x_b);P''_{\mathtt{C}} & \qquad \mathsf{else}\ s[\mathtt{C},\mathtt{S}]!\mathtt{Quit}();\mathbf{0}
\end{array}$$

Note that $P_{\mathtt{C}}$ selects only two of the possible branches (i.e., `Withdraw` and `Quit`) and `Deposit` is never selected. One can think of $P_{\mathtt{C}}$ as an ATM machine that only allows to withdraw a number of £10 banknotes, until the amount exceeds the current balance. This ATM machine does not allow deposits. We assume *getmore*() to be a local function to the principal running $P_{\mathtt{C}}$ that returns tt if more notes are required (ff otherwise). $P_{\mathtt{S}}$ below implements the server role:

$$\begin{array}{l|l}
P_{\mathtt{S}} = s[\mathtt{A},\mathtt{S}]?\{\mathtt{LoginOK}();\mu X.P'_{\mathtt{S}}, \mathtt{LoginFail}().\mathbf{0}\} & P''_{\mathtt{S}} = s[\mathtt{C},\mathtt{S}]?\{\mathtt{Withdraw}(x_p).X, \\
P'_{\mathtt{S}} = s[\mathtt{S},\mathtt{C}]!\mathtt{Account}(\mathit{getBalance}());P''_{\mathtt{S}} & \qquad\qquad \mathtt{Deposit}(x_d).X, \\
 & \qquad\qquad \mathtt{Quit}().\mathbf{0}\ \}
\end{array}$$

where *getBalance*() is a local function to the principal running $P_{\mathtt{S}}$ that synchronously returns the current balance of the client.

3 Theory of Dynamic Safety Assurance

In this section we formalise the specifications (based on local types) used to guard the runtime behaviour of the principals in a network. These specifications can be embedded into system monitors, each wrapping a principal to ensure that the ongoing communication conforms to the given specification. Then, we present a behavioural theory for monitored networks and its safety properties.

3.1 Semantics of Global Specifications

The specification of the (correct) behaviour of a principal consists of an *assertion environment* $\langle \Gamma; \Delta \rangle$, where Γ is the *shared environment* describing the behaviour on shared channels, and Δ is the *session environment* representing the behaviour on session channels (i.e., describing the sessions that the principal is currently participating in). The syntax of Γ and Δ is given by:

$$\Gamma ::= \emptyset \;\big|\; \Gamma, a : \mathtt{I}(T[\mathtt{r}]) \mid \Gamma, a : \mathtt{O}(T[\mathtt{r}]) \qquad \Delta ::= \emptyset \;\big|\; \Delta, s[\mathtt{r}] : T$$

In Γ, the assignment $a : \mathtt{I}(T[\mathtt{r}])$ (resp. $a : \mathtt{O}(T[\mathtt{r}])$) states that the principal can, through a, receive (resp. send) invitations to play role $\mathtt{r}$ in a session instance specified by T. In Δ, we write $s[\mathtt{r}] : T$ when the principal is playing role $\mathtt{r}$ of session s specified by T. Networks are monitored with respect to *collections of specifications* (or just *specifications*) one for each principal in the network. A specification $\Sigma, \Sigma', \ldots$ is a finite map from principals to assertion environments:

$$\Sigma \;::=\; \emptyset \;\big|\; \Sigma, \alpha : \langle \Gamma; \Delta \rangle$$

The semantics of Σ is defined using the following labels:

$$\ell ::= \overline{a}\langle s[\mathtt{r}] : T \rangle \;\big|\; a\langle s[\mathtt{r}] : T \rangle \;\big|\; s[\mathtt{r}_1, \mathtt{r}_2]!l\langle v \rangle \;\big|\; s[\mathtt{r}_1, \mathtt{r}_2]?l\langle v \rangle \;\big|\; \tau$$

The first two labels are for *invitation* actions, the first is for requesting and the second is for accepting. Labels with $s[\mathtt{r}_1, \mathtt{r}_2]$ indicate *interaction* actions for sending (!) or receiving (?) messages within sessions. The labelled transition relation for specification is defined by the rules below.

$$\alpha : \langle \Gamma, a : \mathtt{O}(T[\mathtt{r}]); \Delta \rangle \xrightarrow{\overline{a}\langle s[\mathtt{r}]:T \rangle} \alpha : \langle \Gamma, a : \mathtt{O}(T[\mathtt{r}]); \Delta \rangle \qquad [\text{REQ}]$$

$$\frac{s \notin \mathsf{dom}(\Delta)}{\alpha : \langle \Gamma, a : \mathtt{I}(T[\mathtt{r}]); \Delta \rangle \xrightarrow{a\langle s[\mathtt{r}]:T \rangle} \alpha : \langle \Gamma, a : \mathtt{I}(T[\mathtt{r}]); \Delta, s[\mathtt{r}] : T \rangle} \qquad [\text{ACC}]$$

$$\frac{\Gamma \vdash v : S_j, \quad A_j[v/x_j] \downarrow \mathsf{tt}, \quad j \in I}{\alpha : \langle \Gamma; \Delta, s[\mathtt{r}_2] : \mathtt{r}_1?\{l_i(x_i : S_i)\{A_i\}.T'_i\}_{i \in I} \rangle \xrightarrow{s[\mathtt{r}_1, \mathtt{r}_2]?l_j\langle v \rangle} \alpha : \langle \Gamma; \Delta, s[\mathtt{r}_2] : T'_j[v/x_j] \rangle} \qquad [\text{BRA}]$$

$$\frac{\Gamma \vdash v : S_j, \quad A_j[v/x_j] \downarrow \mathsf{tt}, \quad j \in I}{\alpha : \langle \Gamma; \Delta, s[\mathtt{r}_1] : \mathtt{r}_2!\{l_i(x_i : S_i)\{A_i\}.T'_i\}_{i \in I} \rangle \xrightarrow{s[\mathtt{r}_1, \mathtt{r}_2]!l_j\langle v \rangle} \alpha : \langle \Gamma; \Delta, s[\mathtt{r}_1] : T'_j[v/x_j] \rangle} \qquad [\text{SEL}]$$

$$\frac{\alpha : \langle \Gamma_1; \Delta_1 \rangle \xrightarrow{\ell} \alpha : \langle \Gamma'_1; \Delta'_1 \rangle}{\alpha : \langle \Gamma_1; \Delta_1 | \Delta_2 \rangle \xrightarrow{\ell} \alpha : \langle \Gamma'_1; \Delta'_1 | \Delta_2 \rangle} \qquad \Sigma \xrightarrow{\tau} \Sigma \qquad \frac{\Sigma_1 \xrightarrow{\ell} \Sigma_2}{\Sigma_1, \Sigma_3 \xrightarrow{\ell} \Sigma_2, \Sigma_3} \qquad [\text{SPL,TAU,PAR}]$$

Rule [Req] allows α to send an invitation on a properly typed shared channel a (i.e., given that the shared environment maps a to $T[\mathtt{r}]$). Rule [Acc] allows α to receive an invitation to be role $\mathtt{r}$ in a new session s, on a properly typed shared channel a. Rule [Bra] allows α, participating to sessions s as $\mathtt{r}_2$, to receive a message with label l_j from $\mathtt{r}_1$, given that A_j is satisfied after replacing x_j with the received value v. After the application of this rule the specification is T_j. Rule [Sel] is the symmetric (output) counterpart of [Bra]. We use $\downarrow$ to denote the evaluation of a logical assertion. [Spl] is the parallel composition of two session environments where $\Delta_1|\Delta_2$ composes two local types: $\Delta_1|\Delta_2 = \{s[\mathtt{r}] : (T_1 \mid T_2) \mid T_i = \Delta_i(s[\mathtt{r}]), s[\mathtt{r}] \in \mathsf{dom}(\Delta_1) \cap \mathsf{dom}(\Delta_2)\} \cup \mathsf{dom}(\Delta_1)/\mathsf{dom}(\Delta_2) \cup \mathsf{dom}(\Delta_2)/\mathsf{dom}(\Delta_1)$. [Tau] says that the specification should be invariant under reduction of principals. [Par] says if Σ_1 and Σ_3 are composable, after Σ_1 becomes as Σ_2, they are still composable.

3.2 Semantics of Dynamic Monitoring

The endpoint monitor $\mathsf{M}, \mathsf{M}', \ldots$ for principal α is a specification $\alpha : \langle \Gamma; \Delta \rangle$ used to *dynamically* ensure that the messages to and from α are legal with respect to Γ and Δ. A *monitored network* N is a network N with monitors, obtained by extending the syntax of networks as:

$$\mathsf{N} ::= N \mid \mathsf{M} \mid \mathsf{N} \,|\, \mathsf{N} \mid (\nu s)\mathsf{N} \mid (\nu a)\mathsf{N}$$

The reduction rules for monitored networks are given below and use, in the premises, the labelled transitions of monitors. The labelled transitions of a monitor are the labelled transitions of its corresponding specification (given in § 3.1).

$$\lceil\textsc{Req}\rceil\ \frac{\mathsf{M} \xrightarrow{\overline{a}\langle s[\mathtt{r}]:T\rangle} \mathsf{M}'}{[\overline{a}\langle s[\mathtt{r}] : T\rangle]_\alpha \mid \mathsf{M} \mid \langle r \ ; \ h\rangle \longrightarrow [\mathbf{0}]_\alpha \mid \mathsf{M}' \mid \langle r \ ; \ h \cdot \overline{a}\langle s[\mathtt{r}] : T\rangle\rangle}$$

$$\lceil\textsc{Acc}\rceil\ \frac{\mathsf{M} \xrightarrow{a\langle s[\mathtt{r}]:T\rangle} \mathsf{M}' \qquad r(a) = \alpha}{[a(y[\mathtt{r}] : T).P]_\alpha \mid \mathsf{M} \mid \langle r \ ; \ \overline{a}\langle s[\mathtt{r}] : T\rangle \cdot h\rangle \longrightarrow [P[s/y]]_\alpha \mid \mathsf{M}' \mid \langle r \cdot s[\mathtt{r}] \mapsto \alpha \ ; \ h\rangle}$$

$$\lceil\textsc{Bra}\rceil\ \frac{\mathsf{M} \xrightarrow{s[\mathtt{r}_1,\mathtt{r}_2]?l_j\langle v\rangle} \mathsf{M}' \qquad r(s[\mathtt{r}_2]) = \alpha}{[s[\mathtt{r}_1, \mathtt{r}_2]?\{l_i(x_i).P_i\}_i]_\alpha \mid \mathsf{M} \mid \langle r \ ; \ s\langle \mathtt{r}_1, \mathtt{r}_2, l_j\langle v\rangle\rangle \cdot h\rangle \longrightarrow [P_j[v/x_j]]_\alpha \mid \mathsf{M}' \mid \langle r \ ; \ h\rangle}$$

$$\lceil\textsc{Sel}\rceil\ \frac{\mathsf{M} \xrightarrow{s[\mathtt{r}_1,\mathtt{r}_2]!l\langle v\rangle} \mathsf{M}' \quad r(s[\mathtt{r}_2]) \neq \alpha}{[s[\mathtt{r}_1, \mathtt{r}_2]!l\langle v\rangle]_\alpha \mid \mathsf{M} \mid \langle r \ ; \ h\rangle \longrightarrow [\mathbf{0}]_\alpha \mid \mathsf{M}' \mid \langle r \ ; \ h \cdot s\langle \mathtt{r}_1, \mathtt{r}_2, l\langle v\rangle\rangle\rangle}$$

$$\lceil\textsc{ReqEr}\rceil\ \frac{\mathsf{M} \not\xrightarrow{\overline{a}\langle s[\mathtt{r}]:T\rangle}}{[\overline{a}\langle s[\mathtt{r}] : T\rangle]_\alpha \mid \mathsf{M} \mid \langle r \ ; \ h\rangle \longrightarrow [\mathbf{0}]_\alpha \mid \mathsf{M} \mid \langle r \ ; \ h\rangle}$$

$$\lceil\textsc{AccEr}\rceil\ \frac{\mathsf{M} \not\xrightarrow{a\langle s[\mathtt{r}]:T\rangle}}{[a(y[\mathtt{r}] : T).P]_\alpha \mid \mathsf{M} \mid \langle r \ ; \ \overline{a}\langle s[\mathtt{r}] : T\rangle \cdot h\rangle \longrightarrow [a(y[\mathtt{r}] : T).P]_\alpha \mid \mathsf{M} \mid \langle r \ ; \ h\rangle}$$

$$\lceil\textsc{SelEr}\rceil\ \frac{\mathsf{M} \not\xrightarrow{s[\mathtt{r}_1,\mathtt{r}_2]!l\langle v\rangle}}{[s[\mathtt{r}_1, \mathtt{r}_2]!l\langle v\rangle]_\alpha \mid \mathsf{M} \mid \langle r \ ; \ h\rangle \longrightarrow [\mathbf{0}]_\alpha \mid \mathsf{M} \mid \langle r \ ; \ h\rangle}$$

The first four rules model reductions that are allowed by the monitor (i.e., in the premise). Rule ⌈Req⌉ inserts an invitation in the global queue. Rule ⌈Acc⌉ is symmetric and updates the router so that all messages for role $\mathtt{r}$ in session

s will be routed to α. Similarly, $\lceil$Bra$\rceil$ (resp. $\lceil$Sel$\rceil$) extracts (resp. introduces) messages from (resp. in) the global queue. The error cases for $\lceil$Req$\rceil$ and $\lceil$Sel$\rceil$, namely $\lceil$ReqEr$\rceil$ and $\lceil$SelEr$\rceil$, 'skip' the current action (removing it from the process), do not modify the queue, the router nor the state of the monitor. The error cases for $\lceil$Acc$\rceil$ and $\lceil$Bra$\rceil$, namely $\lceil$AccEr$\rceil$ and $\lceil$BraEr$\rceil$ (the latter omitted for space constraint), do not affect the process, which remains ready to perform the action, and remove the violating message from the queue.

Example 4 (ATM: a monitored network). We illustrate the monitored networks for the ATM scenario, where the routing table is defined as

$$r = a \mapsto \alpha, b \mapsto \beta, c \mapsto \gamma, s[\mathtt{S}] \mapsto \alpha, s[\mathtt{C}] \mapsto \beta, s[\mathtt{A}] \mapsto \gamma$$

We consider the fragment of session where the authentication has occurred, the process of $\mathtt{C}$ (resp. $\mathtt{S}$) is $P'_{\mathtt{C}}$ (resp. $P'_{\mathtt{S}}$) from Example 3, and the process of $\mathtt{A}$ is $\mathbf{0}$.

$$\begin{array}{ll}\mathsf{N}_{\mathtt{S}} = [P'_{\mathtt{S}}]_\alpha \mid \mathsf{M}_{\mathtt{S}} = [s[\mathtt{S},\mathtt{C}]!\ \mathtt{Account}(100); P''_{\mathtt{S}}]_\alpha \mid \mathsf{M}_{\mathtt{S}} & \text{(assuming } \mathit{getBalance}() = 100) \\ \mathsf{N}_{\mathtt{C}} = [P'_{\mathtt{C}}]_\beta \mid \mathsf{M}_{\mathtt{C}} = [s[\mathtt{S},\mathtt{C}]?\ \mathtt{Account}(x_b).P''_{\mathtt{C}}]_\beta \mid \mathsf{M}_{\mathtt{C}} & \\ \mathsf{N}_{\mathtt{A}} = [\mathbf{0}]_\gamma \mid \gamma : \langle c : T_{\mathtt{A}}[\mathtt{A}]\ ;\ s[\mathtt{A}] :\ \mathsf{end}\rangle & \end{array}$$

where $\mathsf{M}_{\mathtt{S}} = \alpha : \langle a :\ T_{\mathtt{S}}[\mathtt{S}]\ ;\ s[\mathtt{S}] :\ \mathtt{C}!\ \mathtt{Account}(x_b : \mathsf{int})\{x_b \geq 0\}.T'_{\mathtt{S}}\rangle$ and $\mathsf{M}_{\mathtt{C}}$ is dual.

$$\begin{array}{l}\mathsf{N}_1 = [s[\mathtt{S},\mathtt{C}]!\ \mathtt{Account}(100); P'_{\mathtt{S}}]_\alpha \mid \mathsf{M}_{\mathtt{S}} \ \mid\ [s[\mathtt{S},\mathtt{C}]?\ \mathtt{Account}(x_b).P'_{\mathtt{C}}]_\beta \mid \mathsf{M}_{\mathtt{C}} \ \mid\ \mathsf{N}_{\mathtt{A}} \ \mid\ \langle r\ ;\ \emptyset\rangle \\ \qquad \longrightarrow\longrightarrow [P'_{\mathtt{S}}]_\alpha \mid \mathsf{M}'_{\mathtt{S}} \mid [P'_{\mathtt{C}}[100/x_b]]_\beta \mid \mathsf{M}'_{\mathtt{C}} \mid \mathsf{N}_{\mathtt{A}} \mid \langle r\ ;\ \emptyset\rangle\end{array}$$

where $\mathsf{M}'_{\mathtt{S}} = \alpha : \langle a{:}T_{\mathtt{S}}[\mathtt{S}]\ ;\ s[\mathtt{S}] :\ T'_{\mathtt{S}}\rangle$ and $\mathsf{M}'_{\mathtt{C}} = \beta : \langle b : T_{\mathtt{C}}[\mathtt{C}]\ ;\ s[\mathtt{C}] :\ T'_{\mathtt{C}}\rangle$

Above, $x_b \geq 0$ is satisfied since $x_b = 100$. If the server tried to communicate e.g., value -100 for x_b, the monitoring (by rule $\lceil$SelEr$\rceil$) would drop the message.

3.3 Network Satisfaction and Equivalences

Based on the formal representations of monitored networks, we now introduce the key formal tools for analysing their behaviour. First, we introduce *bisimulation* and *barbed congruence* over networks, and develop the notion of *interface*. Then we define *the satisfaction relation* $\models N\ :\ \mathsf{M}$, used in § 3.4 to prove the properties of our framework.

Bisimulations. We use $M, M', ...$ for a *partial network*, that is a network which does not contain a global transport, hence enabling the global observation of interactions. The labelled transition relation for processes and partial networks M is defined below.

$$\text{(Req)}\ \ [\overline{a}\langle s[\mathtt{r}] : T\rangle; P]_\alpha \xrightarrow{\overline{a}\langle s[\mathtt{r}]:T\rangle} [\mathbf{0}]_\alpha \qquad \text{(Acc)}\ \ [a(y[\mathtt{r}] : T).P]_\alpha \xrightarrow{a\langle s[\mathtt{r}]:T\rangle} [P[s/y]]_\alpha$$

$$\text{(Bra)}\ \ [s[\mathtt{r}_1,\mathtt{r}_2]?\{l_i(x_i{:}S_i).P_i\}_i]_\alpha \xrightarrow{s[\mathtt{r}_1,\mathtt{r}_2]?l_j\langle v\rangle} [P_j[v/x_j]]_\alpha$$

$$\text{(Sel)}\ \ [s[\mathtt{r}_1,\mathtt{r}_2]!l_j\langle v\rangle]_\alpha \xrightarrow{s[\mathtt{r}_1,\mathtt{r}_2]!l_j\langle v\rangle} [\mathbf{0}]_\alpha \qquad \text{(ctx)}\ \ \frac{[P]_\alpha \xrightarrow{\ell} [P']_\alpha \quad \mathsf{n}(\ell) \cap \mathsf{bn}(\mathcal{E}) = \emptyset}{[\mathcal{E}(P)]_\alpha \xrightarrow{\ell} [\mathcal{E}(P')]_\alpha}$$

$$\text{(tau)}\ \ \frac{M \longrightarrow M'}{M \xrightarrow{\tau} M'} \qquad \text{(res)}\ \ \frac{M \xrightarrow{\ell} M' \quad a \notin \mathsf{sbj}(\ell)}{(\nu a)M \xrightarrow{\ell\backslash a} (\nu a)M'} \qquad \text{(str)}\ \frac{M \equiv M_0 \xrightarrow{\ell} M'_0 \equiv M'}{M \xrightarrow{\ell} M'}$$

In (CTX), $\mathsf{n}(\ell)$ indicates the names occurring in ℓ while $\mathsf{bn}(\mathcal{E})$ indicates binding $\mathcal{E}$ induces. In (RES), $\mathsf{sbj}(\ell)$ denotes the subject of ℓ. In (TAU) the axiom is obtained either from the reduction rules for dynamic networks given in § 2.2 (only those not involving the global transport), or from the corresponding rules for monitored networks (which have been omitted in § 3.2).

Hereafter we write $\Longrightarrow$ for $\xrightarrow{\tau}^{*}$, $\overset{\ell}{\Longrightarrow}$ for $\Longrightarrow\xrightarrow{\ell}\Longrightarrow$, and $\overset{\hat{\ell}}{\Longrightarrow}$ for $\Longrightarrow$ if $\ell = \tau$ and $\overset{\ell}{\Longrightarrow}$ otherwise.

Definition 1 (Bisimulation over partial networks). A binary relation $\mathcal{R}$ over partial networks is a *weak bisimulation* when $M_1 \mathcal{R} M_2$ implies: whenever $M_1 \xrightarrow{\ell} M_1'$ such that $\mathsf{bn}(\ell) \cap \mathsf{fn}(M_2) = \emptyset$, we have $M_2 \overset{\hat{\ell}}{\Longrightarrow} M_2'$ such that $M_1' \mathcal{R} M_2'$, and the symmetric case. We write $M_1 \approx M_2$ if (M_1, M_2) are in a weak bisimulation.

Interface. We want to build a model where two different implementations of the same service are related. Bisimilarity is too strong for this aim (as shown in Example 5). We use instead a contextual congruence (barbed reduction-closed congruence [14]) $\cong$ for networks. Intuitively, two networks are barbed-congruent when they are indistinguishable for any principal that connects to them. In this case we say they propose the same *interface* to the exterior. Formally, two networks are related with $\cong$ when, composed with the same third network, they offer the same *barbs* (the messages to external principals in the respective global queues are on the same channels) and this property is preserved under reduction.

We say that a message m is *routed for α in N* if $N = (\nu \tilde{n})(M_0 \mid \langle r\ ;\ h\rangle)$, $m \in h$, either $m = \overline{a}\langle s[\mathtt{r}] : T\rangle$ and $r(a) = \alpha$ or $m = s[\mathtt{r}_1, \mathtt{r}_2]!l\langle e\rangle$ and $r(s[\mathtt{r}_2]) = \alpha$.

Definition 2 (Barb). We write $N \downarrow_a$ when the global queue of N contains a message m to free a and m is routed for a principal not in N. We write $N \Downarrow_a$ if $N \longrightarrow^* N' \downarrow_a$.

We denote $\mathcal{P}(N)$ for a set of principals in N, $\mathcal{P}(\prod[P_i]_{\alpha_i}) = \{\alpha_1, ..., \alpha_n\}$. We say N_1 and N_2 are *composable* when $\mathcal{P}(N_1) \cap \mathcal{P}(N_2) = \emptyset$, the union of their routing tables remains a function, and their free session names are disjoint. If N_1 and N_2 are composable, we define $N_1 \diamond N_2 = (\nu \tilde{n}_1, \tilde{n}_2)(M_1 \mid M_2 \mid \langle r_1 \cup r_2\ ;\ h_1 \cdot h_2\rangle)$ where $N_i = (\nu \tilde{n}_i)(M_i \mid \langle r_i\ ;\ h_i\rangle)$ $(i = 1, 2)$. Notice that both equivalences are compositional, as proved in Proposition § 4.

Definition 3 (Barbed reduction-closed congruence). A relation $\mathcal{R}$ on networks with the same principals is a *barbed r.c. congruence* [14] if the following holds: whenever $N_1 \mathcal{R} N_2$ we have: (1) for each composable N, $N \diamond N_1 \mathcal{R} N \diamond N_2$; (2) $N_1 \longrightarrow N_1'$ implies $N_2 \longrightarrow^* N_2'$ s.t. $N_1' \mathcal{R} N_2'$ again, and the symmetric case; (3) $N_1 \Downarrow_a$ iff $N_2 \Downarrow_a$. We write $N_1 \cong N_2$ when they are related by a barbed r.c. congruence.

The following result states that composing two bisimilar partial networks with the same network – implying the same router and global transport – yields two undistinguishable networks.

Proposition 4 (Congruency). If $M_1 \approx M_2$, then (1) $M_1|M \approx M_2|M$ for each composable partial M; and (2) $M_1|N \cong M_2|N$ for each composable N.

Example 5 (ATM: an example of behavioural equivalence). We use an example to illustrate our notion of interface. As our verification by monitors is done separately for each endpoint, one can safely modify a global specification as long as its projection on the public roles stays the same. The barbed congruence we introduce takes this into account: two networks proposing the same service, but organised in different ways, are equated even if the two networks correspond to different global specifications. As an example, consider global type $G^2_{\mathtt{ATM}}$ defined as $G_{\mathtt{ATM}}$ where $G^2_{\mathtt{Loop}}$ is used in place of $G_{\mathtt{Loop}}$ from Example 3. $G^2_{\mathtt{Loop}}$ involves a fourth party, the transaction agent B: S sends a query to B which gives back a one-use transaction identifier. Then, the protocol proceeds as the original one. Notably, $G_{\mathtt{ATM}}$ and $G^2_{\mathtt{ATM}}$ have the same interfaces for the client (resp. the authenticator), as their projections of on C (resp. A) are equal.

$$
\begin{array}{ll}
G^2_{\mathtt{Loop}} = \mu\,\mathtt{LOOP}. & \\
\quad \mathtt{S} \to \mathtt{B} : \{ & \mathtt{Query}()\{\mathsf{true}\}. \\
\quad \mathtt{B} \to \mathtt{S} : \{ & \mathtt{Answer}(x_t : \mathsf{int})\{\mathsf{true}\}. \\
\quad \mathtt{S} \to \mathtt{C} : \{ & \mathtt{Account}(x_b : \mathsf{int})\{x_b \geq 0\}. \\
\quad \mathtt{C} \to \mathtt{S} : \{ & \mathtt{Withdraw}(x_p : \mathsf{int})\{x_p \geq 0 \wedge x_b - x_p \geq 0\}.\ \mathtt{LOOP}, \\
 & \mathtt{Deposit}(x_d : \mathsf{int})\{x_d > 0\}.\ \mathtt{LOOP}, \\
 & \mathtt{Quit}()\{\mathsf{true}\}.\mathsf{end} \qquad \}\}\}\}
\end{array}
$$

We define $P^2_{\mathtt{S}}$ as $P_{\mathtt{S}}$ in Example 3 but replacing the occurrence of $P'_{\mathtt{S}}$ in $P_{\mathtt{S}}$ by

$$s[\mathtt{S},\mathtt{B}]!\mathtt{Query}\langle\rangle; s[\mathtt{B},\mathtt{S}]?\mathtt{Answer}(x_t).P'_{\mathtt{S}}$$

and also $N^2_{\mathtt{S}} = [P^2_{\mathtt{S}}]_\alpha$ and $N_{\mathtt{B}} = [\mu X.s[\mathtt{S},\mathtt{B}]?\mathtt{Query}\langle\rangle; s[\mathtt{B},\mathtt{S}]!\mathtt{Answer}\langle getTrans()\rangle]_\delta$. By definition, the two following networks are barbed-congruent:

$$
\begin{array}{l}
(N_{\mathtt{S}} \mid \langle \emptyset\ ;\ s[\mathtt{S}] \mapsto \alpha, s[\mathtt{C}] \mapsto \beta, s[\mathtt{A}] \mapsto \gamma\rangle) \cong \\
\qquad (N^2_{\mathtt{S}} \mid N_{\mathtt{B}} \mid \langle \emptyset\ ;\ s[\mathtt{S}] \mapsto \alpha, s[\mathtt{C}] \mapsto \beta, s[\mathtt{A}] \mapsto \gamma, s[\mathtt{B}] \mapsto \delta\rangle)
\end{array}
$$

even if the first one implements the original ATM protocol while the second one implements its variant. Indeed, composed with any tester, such as $N_{\mathtt{C}} \mid N_{\mathtt{A}} = [P_{\mathtt{C}}]_\beta \mid [P_{\mathtt{A}}]_\gamma$ these two networks will produce the same interactions.

However, the corresponding partial networks $N^2_{\mathtt{S}} \mid N_{\mathtt{B}}$ and $N_{\mathtt{S}}$ are not bisimilar: the former is able to perform a transition labelled $s[\mathtt{S},\mathtt{B}]!\mathtt{Query}\langle\rangle$ while the latter is not. This difference in behaviour is not visible to the barbed congruence, as it takes into account the router which prevents the messages $s[\mathtt{S},\mathtt{B}]!\mathtt{Query}\langle\rangle$ to be caught by a tester. As an example of network bisimilar to $N_{\mathtt{S}}$, consider:

$$N_1 = (\nu k)\ ([P_{\mathtt{S}} \mid P_{\mathtt{S}}[k/s]]_\alpha \ \mid\ [P_{\mathtt{C}}[k/s]]_\delta)$$

In this partial network, principal α plays both S in public session s (as in $N_{\mathtt{S}}$) and S in the private session k. Principal δ plays C in the latter. As k is private, N_1 offers the same observable behaviour than $N_{\mathtt{S}}$ (no action on k can be observed), and we have $N_1 \approx N_{\mathtt{S}}$.

Satisfaction. We present a satisfaction relation for partial networks, which include local principals. If M is a partial network, $\models M : \Sigma$ s.t. $\mathsf{dom}(\Sigma) = \mathcal{P}(M)$, means that the specification allows all outputs from the network; that the network is ready to receive all the inputs indicated by the specification; and that this is preserved by transition.

Definition 5 (Satisfaction). Let $\mathsf{sbj}(\ell)$ denote the subject of $\ell \neq \tau$. A relation $\mathcal{R}$ from partial networks to specifications is a *satisfaction* when $M\mathcal{R}\Sigma$ implies:

1. If $\Sigma \xrightarrow{\ell} \Sigma'$ for an input ℓ and M has an input at $\mathsf{sbj}(\ell)$, then $M \xrightarrow{\ell} M'$ s.t. $M'\mathcal{R}\Sigma'$.
2. If $M \xrightarrow{\ell} M'$ for an output at ℓ, then $\Sigma \xrightarrow{\ell} \Sigma'$ s.t. $M'\mathcal{R}\Sigma'$.
3. If $M \xrightarrow{\tau} M'$, then $\Sigma \xrightarrow{\tau} \Sigma'$ s.t. $M'\mathcal{R}\Sigma'$ (i.e. $M'\mathcal{R}\Sigma$ since $\Sigma \xrightarrow{\tau} \Sigma$ always).

When $M\mathcal{R}\Sigma$ for a satisfaction relation $\mathcal{R}$, we say M *satisfies* Σ, denoted $\models M : \Sigma$. By Definition 5 and Proposition 4 we obtain:

Proposition 6. If $M_1 \cong M_2$ and $\models M_1 : \Sigma$ then $\models M_2 : \Sigma$.

3.4 Safety Assurance and Session Fidelity

In this section, we present the properties underpinning safety assurance in the proposed framework from different perspectives.

Theorem 7 shows local safety/transparency, and global safety/transparency for fully monitored networks. A network N is *fully monitored wrt* Σ when all its principals are monitored and the collection of the monitors is congruent to Σ.

Theorem 7 (Safety and Transparency)

1. **(Local Safety)** $\models [P]_\alpha \mid \mathsf{M} : \alpha : \langle \Gamma; \Delta \rangle$ with $\mathsf{M} = \alpha : \langle \Gamma; \Delta \rangle$.
2. **(Local Transparency)** If $\models [P]_\alpha : \alpha : \langle \Gamma; \Delta \rangle$, then $[P]_\alpha \approx ([P]_\alpha \mid \mathsf{M})$ with $\mathsf{M} = \alpha : \langle \Gamma; \Delta \rangle$.
3. **(Global Safety)** If N is fully monitored w.r.t. Σ, then $\models \mathsf{N} : \Sigma$.
4. **(Global Transparency)** Assum N and N have the same global transport $\langle r\ ;\ h \rangle$. If N is fully monitored w.r.t. Σ and $N = M \mid \langle r\ ;\ h \rangle$ is unmonitored but $\models M : \Sigma$, then we have $\mathsf{N} \sim N$.

Local safety (7.1) states that a monitored process always behaves well with respect to the specification. Local transparency (7.2) states that a monitored process behaves as an unmonitored process when the latter is well-behaved (e.g., it is statically checked). Global safety (7.3) states that a fully monitored network behaves well with respect to the given global specification. This property is closely related to session fidelity, introduced later in Theorem 11. Global transparency (7.4) states that a monitored network and an unmonitored network have equivalent behaviour when the latter is well-behaved with respect to the same (collection of) specifications.

By Proposition 4 and (7.2), we derive Corollary 8 stating that weakly bisimilar static networks combined with the same global transport are congruent.

Corollary 8 (Local transparency). If $\models [P]_\alpha : \alpha : \langle \Gamma; \Delta \rangle$, then for any $\langle r\ ;\ h \rangle$, we have $([P]_\alpha \mid \langle r\ ;\ h \rangle) \cong ([P]_\alpha \mid \mathsf{M} \mid \langle r\ ;\ h \rangle)$ with $\mathsf{M} = \alpha : \langle \Gamma; \Delta \rangle$.

By Theorem 7, we can *mix* unmonitored principals with monitored principals still obtaining the desired safety assurances.

In the following, we refer to a pair $\Sigma; \langle r \ ; \ h\rangle$ of a specification and a global transport as a *configuration*. The labelled transition relation for configurations, denoted by $\xrightarrow{\ell}_{\mathrm{g}}$, is relegated to [2]. Here it is sufficient to notice that the transitions of a configuration define the correct behaviours (with respect to Σ) in terms of the observation of inputs and outputs from/to the global transport $\langle r \ ; \ h\rangle$. We write that a configuration $\Sigma; \langle r \ ; \ h\rangle$ is *configurationally consistent* if all of its multi-step input transition derivatives are receivable and the resulting specifications Σ is consistent.

We also use $\xrightarrow{\ell}_{\mathrm{g}}$ to model globally visible transitions of networks (i.e., those locally visible transitions of a network that can be observed by its global transport). Below, we state that a message emitted by a valid output action is always receivable.

Lemma 9. Assume a network $N \equiv M|\langle r \ ; \ h\rangle$ conforming to $\Sigma; \langle r \ ; \ h\rangle$ which is configurationally consistent, if $N \xrightarrow{\ell}_{\mathrm{g}} N'$ such that ℓ is an output and $\Sigma; \langle r \ ; \ h\rangle \xrightarrow{\ell}_{\mathrm{g}} \Sigma'; \langle r \ ; \ h{\cdot}m\rangle$ then $h \cdot m$ is receivable to Σ'.

Also, we state that, as $N \equiv M \mid H$ and $\models M \ : \ \Sigma$, the satisfaction relation of M and Σ is preserved by transitions.

Lemma 10. Assume $N \equiv M \mid H$ and $\models M \ : \ \Sigma$. If $N \xrightarrow{\ell}_{\mathrm{g}} N' \equiv M' \mid H'$ and $\Sigma \xrightarrow{\ell} \Sigma'$, then $\models M' \ : \ \Sigma'$.

Theorem 11 (Session Fidelity). Assume configuration $\Sigma; \langle r \ ; \ h\rangle$ is configurationally consistent, and network $N \equiv M|\langle r \ ; \ h\rangle$ conforms to configuration $\Sigma; \langle r \ ; \ h\rangle$. For any ℓ, whenever we have $N \xrightarrow{\ell}_{\mathrm{g}} N'$ s.t. $\Sigma; \langle r \ ; \ h\rangle \xrightarrow{\ell}_{\mathrm{g}} \Sigma'; \langle r' \ ; \ h'\rangle$, it holds that $\Sigma'; \langle r' \ ; \ h'\rangle$ is configurationally consistent and N' conforms to $\Sigma'; \langle r' \ ; \ h'\rangle$.

By session fidelity, if all session message exchanges in a monitored/unmonitored network behave well with respect to the specifications (as communications occur), then this network exactly follows the original global specifications.

4 Conclusion and Future Work

We proposed a new formal safety assurance framework to specify and enforce the global safety for distributed systems, through the use of static and dynamic verification. We formally proved the correctness (with respect to distributed principals) of our architectural framework through a π-calculus based theory, identified in two key properties of dynamic network: global transparency and safety. We introduced a behavioural theory over monitored networks which allows compositional reasoning over trusted and untrusted (but monitored) components.

Implementation. As a part of our collaboration with the Ocean Observatories Initiative [17], our theoretical framework is currently realised by an implementation, in which each monitor supports all well-formed protocols and is automatically self-configured, via session initiation messages, for all sessions that the endpoint participates in. Our implementation of the framework automates distributed monitoring by generating FSM from the local protocol projections. In this implementation, the global protocol serves as the key abstraction that helps unify the aspects of specification, implementation and verification (both static and dynamic) of distributed application development. Our experience has shown that the specification framework can accommodate diverse practical use cases, including real-world communication patterns used in the distributed services of the OOI cyberinfrastructure [17].

Future Work. Our objectives include the incorporation in the implementation of more elaborate handling of error cases into monitor functionality, such as halting all local sessions or coercing to valid actions [18,16]. In order to reach this goal, we need to combine a simplification of [5] and nested sessions [10] to handle an exception inside MPSTs. We aim to construct a simple and reliable way to raise and catch exceptions in asynchronous networks. Our work is motivated by ongoing collaborations with the Savara[2] and Scribble[3] projects and OOI [17]. We are continuing the development of Scribble, its toolsuite and associated environments towards a full integration of sessions into the OOI infrastructure.

4.1 Related Work

Our work features a located, distributed process calculus to model monitored networks. Due to space limitations, we focus on the key differences with related work on dynamic monitoring.

The work in [13] proposes an ambient-based run-time monitoring formalism, called guardians, targeted at access control rights for network processes, and Klaim [9] advocates a hybrid (dynamic and static) approach for access control against capabilities (policies) to support static checking integrated within a dynamic access-control procedure. These works address specific forms of access control for mobility, while our more general approach aims at ensuring correct behaviour in sessions through a combination of static or run-time verification.

The work in [4] presents a monitor-based information-flow analysis in multiparty sessions. The monitors in [4] are inline (according to [6]) and control the information-flow by tagging each message with security levels. Our monitors are outline and aim at the application to distributed systems.

An informal approach to monitoring based on MPSTs, and an outline of monitors are presented in [8]. However, [8] only gives an overview of the desired properties, and requires all local processes to be dynamically verified through the protections of system monitors. In this paper, instead, we integrate statically and dynamically verified local processes into one network, and formally state the properties of this combination.

In summary, compared to these related works, our contribution focuses on the enforcement of global safety, with protocols specified as multiparty session types

[2] `http://www.jboss.org/savara`

[3] `http://www.scribble.org`

with assertions. It also provides formalisms and theorems for decentralised run-time monitoring, targeting interaction between components written in multiple (e.g., statically and dynamically typed) programming languages.

References

1. Bettini, L., Coppo, M., D'Antoni, L., De Luca, M., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)
2. Bocchi, L., Chen, T.-C., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. Technical Report 2013/3, Department of Computing, Imperial College London (2013)
3. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 162–176. Springer, Heidelberg (2010)
4. Capecchi, S., Castellani, I., Dezani-Ciancaglini, M.: Information flow safety in multiparty sessions. In: EXPRESS. EPTCS, vol. 64, pp. 16–30 (2011)
5. Capecchi, S., Giachino, E., Yoshida, N.: Global escape in multiparty session. In: FSTTCS 2010. LIPICS, vol. 8, pp. 338–351 (2010)
6. Chen, F., Rosu, G.: MOP:An Efficient and Generic Runtime Verification Framework. In: OOPSLA, pp. 569–588 (2007)
7. Chen, T.-C.: Theories for Session-based Governance for Large-Scale Distributed Systems. PhD thesis, Queen Mary, University of London (to appear, 2013)
8. Chen, T.-C., Bocchi, L., Deniélou, P.-M., Honda, K., Yoshida, N.: Asynchronous distributed monitoring for multiparty session enforcement. In: Bruni, R., Sassone, V. (eds.) TGC 2011. LNCS, vol. 7173, pp. 25–45. Springer, Heidelberg (2012)
9. De Nicola, R., Ferrari, G., Pugliese, R.: Klaim: a kernel language for agents interaction and mobility. IEEE Trans. Softw. Eng. 24, 315–330 (1998)
10. Demangeon, R., Honda, K.: Nested protocols in session types. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 272–286. Springer, Heidelberg (2012)
11. Deniélou, P.-M., Yoshida, N.: Dynamic multirole session types. In: POPL, pp. 435–446 (2011)
12. Deniélou, P.-M., Yoshida, N.: Multiparty session types meet communicating automata. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 194–213. Springer, Heidelberg (2012)
13. Ferrari, G., Moggi, E., Pugliese, R.: Guardians for ambient-based monitoring. In: F-WAN, pp. 141–202. Elsevier (2002)
14. Honda, K., Yoshida, N.: On reduction-based process semantics. TCS 151(2), 437–486 (1995)
15. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL 2008, pp. 273–284. ACM (2008)
16. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Trans. Inf. Syst. Secur. 12, 19:1–19:41 (2009)
17. OOI, `http://www.oceanobservatories.org/`
18. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. 3, 30–50 (2000)
19. Yoshida, N., Deniélou, P.-M., Bejleri, A., Hu, R.: Parameterised multiparty session types. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 128–145. Springer, Heidelberg (2010)

Semantic Subtyping for Objects and Classes

Ornela Dardha[1], Daniele Gorla[2], and Daniele Varacca[3]

[1] INRIA Focus Team / Università di Bologna, Italy
[2] Dip. di Informatica, "Sapienza" Università di Roma, Italy
[3] PPS - Université Paris Diderot & CNRS, France

Abstract. We propose an integration of structural subtyping with boolean connectives and semantic subtyping to define a Java-like programming language that exploits the benefits of both techniques. Semantic subtyping is an approach to defining subtyping relation based on set-theoretic models, rather than syntactic rules. On the one hand, this approach involves some non trivial mathematical machinery in the background. On the other hand, final users of the language need not know this machinery and the resulting subtyping relation is very powerful and intuitive. While semantic subtyping is naturally linked to the structural one, we show how the framework can also accommodate the nominal subtyping. Several examples show the expressivity and the practical advantages of our proposal.

1 Introduction

Type systems for programming languages are often based on a subtyping relation on types. There are two main approaches for defining the subtyping relation: the *syntactic* approach and the *semantic* one. The syntactic approach is more common: the subtyping relation is defined by means of a formal system of deductive rules. One proceeds as follows: first define the language, then the set of syntactic types and finally the subtyping relation by inference rules. In the semantic approach, instead, one starts from a model of the language and an interpretation of types as subsets of the model. The subtyping relation is then defined as inclusion of sets denoting types. This approach has received less attention than the syntactic one as it is more technical and constraining: it is not trivial to define the interpretation of types as subsets of a model. On the other hand, it presents several advantages: it allows a natural definition of boolean operators, also the meaning of types is more intuitive for the programmer, who need not be aware of the theory behind the curtain.

The first use of the semantic approach goes back to two decades ago [3,10]. More recently, Hosoya and Pierce [15,16] have adopted this approach to define XDuce, an XML-oriented language which transforms XML documents into other XML documents, satisfying certain properties. Subtyping relation is established as inclusion of sets of values, the latter being fragments of XML documents. Castagna et al [8,13] extend the XDuce with first-class functions and arrow types defining a higher-order language, named $\mathbb{C}$Duce, and adopting the semantic approach to subtyping. The starting point of their framework is a higher-order $\lambda-$calculus with pairs and projections. The set of types is extended with intersection, union and negation types interpreted in a set-theoretic way. This approach can also be applied to π-calculus [25]. Castagna et al. [9] defined the $\mathbb{C}\pi$ language, a variant of the asynchronous π-calculus where channel types are augmented with boolean connectives. Finally, semantic subtyping is adopted in a flow-typing calculus [23]. Flow-typing allows a variable to have different types in different parts of a program and thus is more flexible than the standard static typing. Type systems for flow-typing incorporate intersection, union and negation types in order to typecheck terms, like for example, *if-then-else* statements. Consequently, semantic subtyping is naturally defined on top of these systems.

In the present paper, we address the semantic subtyping approach by applying it to an object-oriented core language. Our starting point is *Featherweight Java* (FJ) [17], which is a functional fragment of Java. From a technical point of view the development is challenging. It follows [13], but with the difference that we do not have higher-order values. Therefore, we cannot directly reuse their results. Instead, we define from scratch the semantic model that induces the subtyping relation, and we prove some important theoretical results. The mathematical technicalities, however, are transparent to the final user. Thus, the overheads are hidden to the programmer. The benefits reside in that the programmer now has a language with no additional complexity (w.r.t. standard Java) but with an easier-to-write, more expressive set of types. There are several other reasons and benefits that make semantic subtyping very appealing in an object-oriented setting. For example, it allows us to very easily handle powerful *boolean type constructors* and model both *structural* and *nominal* subtyping. The importance, both from the theoretical and the practical side, of boolean type constructors is widely known in several settings, e.g. in the λ-calculus. Below, we show two examples where the advantages of using boolean connectives in an object-oriented language become apparent.

Boolean Constructors for Modeling Multimethods. Featherweight Java [17] is a core language, so several features of full Java are not included in it; in particular, an important missing feature is the possibility of overloading methods, both in the same class or along the class hierarchy. By using boolean constructors, the type of an overloaded method can be expressed in a very compact and elegant way, and this modeling comes for free after having defined the semantic subtyping machinery. Actually, what we are going to model is not Java's overloading (where the *static* type of the argument is considered for resolving method invocations) but *multimethods* (where the *dynamic* type is considered). To be precise, we implement the form of multimethods used, e.g., in [5,7]; according to [6], this form of multimethods is "very clean and easy to understand [...] it would be the best solution for a brand new language".

As an example, consider the following class declarations:[1]

```
class A extends Object {                class B extends A {
        ...                                     ...
        int length (string s){ ... }            int length (int n){ ... }
}                                       }
```

As expected, method *length* of A has type **string** $\rightarrow$ **int**. However, such a method in B has type (**string** $\rightarrow$ **int**)$\wedge$(**int** $\rightarrow$ **int**),[2] which can be simplified as (**string**$\vee$ **int**) $\rightarrow$ **int**.

The Use of Negation Types. Negation types can be used by the compiler for typechecking terms of a language. But they can also be used directly by the programmer. Suppose we want to represent an inhabitant of Christiania, that does not want to use money and does not want to deal with anything that can be given a price. In this scenario, we have a collection of objects, some of which may have a *getValue* method that tells their value in €. We want to implement a class *Hippy* which has a method *barter* that is intended to be applied only to objects that do not have the method *getValue*. This is very difficult to represent in a language with only nominal subtyping; but also in a language with structural subtyping, it is not clear how to express the fact that a method is not present.

In our framework we offer an elegant solution by assigning to objects that have the method *getValue* the type denoted by

$$[\mathit{getValue} : \mathbf{void} \rightarrow \mathbf{real}]$$

Within the class *Hippy*, we can now define a method with signature

$$\mathbf{void}\ \mathit{barter}(\neg[\mathit{getValue} : \mathbf{void} \rightarrow \mathbf{real}]\ x)$$

that takes in input only objects x that do not have a price, i.e., a method named *getValue*. One could argue that it is difficult to statically know that an object does not have method *getValue* and thus no reasonable application of method *barter* can be well typed. However, it is not difficult to explicitly build a collection of objects that do not have method *getValue*, by dynamically checking the presence of the method. This is possible thanks to the **instanceof** construction (described in Section 5.3). Method *barter* can now be applied to any object of that list, and the application will be well typed.

In the case of a language with nominal subtyping, one can enforce the policy that objects with a price implement the interface *ValuedObject*. Then, the method *barter* would take as input only objects of type $\neg$ *ValuedObject*.

While the example is quite simple, we believe it exemplifies the situations in which we want to statically refer to a portion of a given class hierarchy and

[1] Here and in the rest of the paper we use '...' to avoid writing the useless part of a class, e.g. constructors or irrelevant fields/methods.

[2] To be precise, the actual type is ((**string**$\wedge\neg$**int**) $\rightarrow$ **int**) $\wedge$ (**int** $\rightarrow$ **int**) but **string**$\wedge$ $\neg$**int** $\simeq$ **string**, where $\simeq$ denotes $\leq \cap \leq^{-1}$ and $\leq$ is the (semantic) subtyping relation.

exclude the remainder. The approach we propose is more elegant and straightforward than the classical solution offered by an object-oriented paradigm.

Structural Subtyping. An orthogonal issue, typical of object-oriented languages, is the *nominal* vs. *structural* subtyping question. In a language where subtyping is nominal, A is a subtype of B if and only if it is declared to be so, meaning if class A extends (or implements) class (or interface) B; these relations must be defined by the programmer and are based on the names of classes and interfaces declared. Java programmers are used to nominal subtyping, but other languages [12,14,18,19,20,22,24] are based on the structural approach. In the latter, subtyping relation is established only by analyzing the structure of a class, i.e. its fields and methods: a class A is a subtype of a class B if and only if the fields and methods of A are a superset of the fields and methods of B, and their types in A are subtypes of their types in B. Even though the syntactic subtyping is more naturally linked to the nominal one, the former can also be adapted to support the structural one, as shown in [14,19]. In this paper we follow the reverse direction and give another contribution. The definition of structural subtyping as inclusion of sets fits perfectly the definition of semantic subtyping. So, we integrate both approaches in the same framework. In addition to that, with minor modifications, it is also possible to include in the framework the choice of using nominal subtyping without changing the underlying theory. Thus, since both nominal and structural subtyping are thoroughly used and have their benefits, in our work, we can have them both and so it becomes a programmer's decision on what subtyping to adopt.

Plan of the Paper. In Section 2 we present the syntax of types and terms. In Section 3 we define type models, semantic subtyping relation and present also the typing rules. In Section 4 we present the operational semantics and the soundness of the type system. Proofs of theorems and auxiliary lemmas can be found in [11]. Section 5 gives a discussion on the calculus and Section 6 concludes the paper.

2 The Calculus

In this section, we present the syntax of the calculus, starting first with the types and then the language terms, which are substantially the ones in FJ.

2.1 Types

Our types are defined by properly restricting the type terms inductively defined by the following grammar:

$$\begin{array}{rll} \tau ::= & \alpha \mid \mu & \textit{Type term} \\ \alpha ::= & \mathbf{0} \mid \mathbb{B} \mid [\widetilde{l : \tau}] \mid \alpha \wedge \alpha \mid \neg\alpha & \textit{Object type}\ (\alpha\text{-type}) \\ \mu ::= & \alpha \to \alpha \mid \mu \wedge \mu \mid \neg\mu & \textit{Method type}\ (\mu\text{-type}) \end{array}$$

Types can be of two kinds: α-types (used for declaring fields and, in particular, objects) and μ-types (used for declaring methods). Arrow types are needed to type the methods of our calculus. Since our language is first-order and methods are not first-class values, arrow types are introduced by a separate syntactic category, viz. μ. Type $\mathbf{0}$ denotes the empty type. Type $\mathbb{B}$ denotes the basic types: integers, booleans, void, etc. Type $[\widetilde{l:\tau}]$ denotes a record type, where $\widetilde{l:\tau}$ indicates the sequence $l_1 : \tau_1, \ldots, l_k : \tau_k$, for some $k \geq 0$. Labels l range over an infinite countable set $\mathcal{L}$. When necessary, we will write a record type as $[\widetilde{a:\alpha}, \widetilde{m:\mu}]$ to emphasize the fields of the record, denoted by the labels $\widetilde{a}$, and the methods of the record, denoted by $\widetilde{m}$. Given a type $\rho = [\widetilde{a:\alpha}, \widetilde{m:\mu}]$, $\rho(a_i)$ is the type assigned to the field a_i and $\rho(m_j)$ is the type assigned to the method m_j. In each record type $a_i \neq a_j$ for $i \neq j$ and $m_h \neq m_k$ for $h \neq k$. To simplify the presentation, we are modeling a form of multimethods where at most one definition for every method name is present in every class. However, the general form of multimethods can be recovered by exploiting the simple encoding of Section 5.2. The boolean connectives $\wedge$ and $\neg$ have their intuitive set-theoretic meaning. We use $\mathbf{1}$ to denote the type $\neg\mathbf{0}$ that corresponds to the universal type. We use the abbreviation $\alpha \backslash \alpha'$ to denote $\alpha \wedge \neg\alpha'$ and $\alpha \vee \alpha'$ to denote $\neg(\neg\alpha \wedge \neg\alpha')$. The same holds for the μ-types.

Definition 1 (Types). *The* pre-types *are the regular trees (i.e., the trees with a finite number of non-isomorphic subtrees) produced by the syntax of type terms.*

The set of types*, denoted by $\mathcal{T}$, is the largest set of* well-formed *pre-types, i.e. the ones for which the binary relation $\triangleright$ defined as*

$$\tau_1 \wedge \tau_2 \triangleright \tau_1 \qquad \tau_1 \wedge \tau_2 \triangleright \tau_2 \qquad \neg\tau \triangleright \tau$$

does not contain infinite chains.

Notice that every finite tree obtained by the grammar of types is both regular and well formed; so, it is a type. Problems can arise with infinite trees, which leads us to restrict them to the regular and the well-formed ones. Indeed, if a tree is not-regular, then it is difficult to write it down in a finite way; since we want our types to be usable in practice, we require regular trees that can be easily written down, e.g. by using recursive type equations. Moreover, as we want types to denote sets, we impose some restrictions to avoid ill-formed types. For example, the solution to $\alpha = \alpha \wedge \alpha$ contains no information about the set denoted by α; or $\alpha = \neg\alpha$ does not admit any solution. Such situations are problematic when we define the model. To rule them out, we only consider infinite trees whose branches always contain an atom, where *atoms* are the basic types $\mathbb{B}$, the record types $[\widetilde{l:\tau}]$ and the arrow types $\alpha \to \alpha$. This intuition is what the definition of relation $\triangleright$ formalizes. The restriction to well-formed types is required to avoid meaningless types; the same choice is used in [13]. A different restriction, called *contractiveness*, is used for instance in [4], where non-regular types are also allowed.

2.2 Terms

Our calculus is based on FJ [17] rather than, for example, the object-oriented calculus in [1], because of the widespread diffusion of Java. There is a correspondence between FJ and the pure functional fragment of Java, in a sense that any program in FJ is an executable program in Java. Our syntax is essentially the same as [17], apart from the absence of the *cast* construct and the presence of the **rnd** primitive. We have left out the first construct for the sake of simplicity; it is orthogonal to the aim of the current work and it can be added to the language without any major issue. The second construct is a nondeterministic choice operator. This operator is technically necessary to obtain a completeness result. Indeed, we interpret method types not as function but as relations, following the same line of [13]; thus, a nondeterministic construct is needed to account for this feature. In addition, **rnd** can be used to model side-effects. We assume a countable set of names, among which there are some key names: *Object* that indicates the root class, **this** that indicates the current object, and **super** that indicates the parent object. We will use the letters $A, B, C, \ldots$ for indicating classes, $a, b, \ldots$ for fields, $m, n, \ldots$ for methods and $x, y, z, \ldots$ for variables. $\mathcal{K}$ will denote the set of constants of the language and we will use the meta-variable c to range over $\mathcal{K}$. Generally, to make examples clearer, we will use mnemonic names to indicate classes, methods, etc.; for example, *Point*, *print*, etc.

The syntax of the language is given by the following grammar:

Class declaration	L	$::= \textbf{class}\ C\ \textbf{extends}\ C\ \{\ \widetilde{\alpha\ a};\ K;\ \widetilde{M}\ \}$
Constructor	K	$::= C\ (\widetilde{\beta\ b}; \widetilde{\alpha\ a})\ \{\ \textbf{super}(\widetilde{b});\ \widetilde{\textbf{this}.a} = \widetilde{a};\ \}$
Method declaration	M	$::= \alpha\ m\ (\alpha\ a)\ \{\ \textbf{return}\ e;\ \}$
Expressions	e	$::= x \mid c \mid e.a \mid e.m(e) \mid \textbf{new}\ C(\widetilde{e}) \mid \textbf{rnd}(\alpha)$

A *program* is a pair $(\widetilde{L}, e)$ consisting of a sequence of class declarations (inducing a class hierarchy, as specified by the inheritance relation) $\widetilde{L}$ where the expression e is evaluated. A class declaration L provides the name of the class, the name of the parent class it extends, its fields (each equipped with a type specification), the constructor K and its method declarations M. The constructor initializes the fields of the object by assigning values to the fields inherited by the super-class and to the fields declared in the present class. A method is declared by specifying the return type, the name of the method, the formal parameter (made up by a type specification given to a symbolic name) and a return expression, i.e. the body of the method. For simplicity, we use unary methods without compromising the expressivity of the language: passing tuples of arguments can be modeled by passing an object that instantiates a class, defined in an ad-hoc way for having as fields all the arguments needed. On the other hand, we exploit this simplification in the theoretical development of our framework. Finally, expressions e are variables, constants, field accesses, method invocations, object creations and random choices among values of a given type. In this work we assume that $\widetilde{L}$ is well-defined, in the sense that "it is not possible that a class A extends a class

B and class B extends class A", or "a constructor called B cannot be declared in a class A" and other obvious rules like these. All these kinds of checks could be carried out in the type system, but we prefer to assume them to focus our attention on the new features of our framework. The same sanity checks are assumed also in FJ [17].

3 Semantic Subtyping

3.1 Models

Having defined the raw syntax, we should now introduce the typing rules. They would typically involve a subsumption rule, that invokes a notion of subtyping. It is therefore necessary to define subtyping, As we have already said, in the semantic approach τ_1 is a subtype of τ_2 if all the τ_1-values are also τ_2-values, i.e., if the set of values of type τ_1 is a subset of the set of values of type τ_2. However, in this way, subtyping is defined by relying on the notion of well-typed values; hence, we need the typing relation to determine typing judgments for values; but the typing rules use the subtyping relation which we are going to define. So, there is a circularity. To break this circle, we follow the path of [13] and adapt it to our framework. The idea is to first interpret types as subsets of some abstract "model" and then establish subtyping as set-inclusion. By using this abstract notion of subtyping, we can then define the typing rules. Having now a notion of well-typed value, we can define the "real" interpretation of types as sets of values. This interpretation can be used to define another notion of subtyping. But if the abstract model is chosen carefully, then the real subtyping relation coincides with the abstract one, and the circle is closed. A model consists of a set D and an interpretation function $[\![\_]\!]_D : \mathcal{T} \to \mathcal{P}(D)$. Such a function should interpret boolean connectives in the expected way (conjunction corresponds to intersection and negation corresponds to complement) and should capture the meaning of type constructors. Notice that there can be several models and it is not guaranteed that they all induces the same subtyping relation. For our purposes, we only need to find one suitable model that we shall call *bootstrap model* $[\![\_]\!]_{\mathcal{B}}$. The construction of this model is beyond the scope of this paper, and for a detailed presentation we refer the reader to [11]. Then, set inclusion in the bootstrap model induces a subtyping relation: $\tau_1 \leq_{\mathcal{B}} \tau_2 \iff [\![\tau_1]\!]_{\mathcal{B}} \subseteq [\![\tau_2]\!]_{\mathcal{B}}$.

3.2 Typing Terms

In the typing rules for our language we use the subtyping relation just defined to derive typing judgments $\Gamma \vdash_{\mathcal{B}} e : \tau$. In particular, this means to use $\leq_{\mathcal{B}}$ in the subsumption rule. In the following we just write $\leq$ instead of $\leq_{\mathcal{B}}$. Let us assume a sequence of class declarations $\widetilde{L}$. First of all, we have to determine the (structural) type of every class C in $\widetilde{L}$. To this aim, we have to take into account the inheritance relation specified in the class declarations in $\widetilde{L}$. We write "$a \in C$" to mean that there is a field declaration of name a in class C within the

Table 1. Definition of function *type*(_)

- $type(Object) = [\,]$;
- $type(C) = \rho$, provided that:
  - C extends D in $\widetilde{L}$;
  - $type(D) = \rho'$;
  - for any field name a
    - if $\rho'(a)$ is undefined and $a \notin C$, then $\rho(a)$ is undefined;
    - if $\rho'(a)$ is undefined and $a \in C$ with type α'', then $\rho(a) = \alpha''$;
    - if $\rho'(a)$ is defined and $a \notin C$, then $\rho(a) = \rho'(a)$;
    - if $\rho'(a)$ is defined, $a \in C$ with type α'' and $\alpha'' \leq \rho'(a)$, then $\rho(a) = \alpha''$.

    We assume that all the fields defined in ρ' and not declared in C appear at the beginning of ρ, having the same order as in ρ'; the fields declared in C then follow, respecting their declaration order in C.
  - for any method name m:
    - if $\rho'(m)$ is undefined and $m \notin C$, then $\rho(m)$ is undefined;
    - if $\rho'(m)$ is undefined and $m \in C$ with type $\alpha \to \beta$, then $\rho(m) = \alpha \to \beta$;
    - if $\rho'(m)$ is defined and $m \notin C$, then $\rho(m) = \rho'(m)$;
    - if $\rho'(m) = \bigwedge_{i=1}^{n} \alpha_i \to \beta_i$, $m \in C$ with type $\alpha \to \beta$ and $\mu = \alpha \to \beta \wedge \bigwedge_{i=1}^{n} \alpha_i \setminus \alpha \to \beta_i \leq \rho'(m)$, then $\rho(m) = \mu$.

  $type(C)$ is undefined, otherwise.

hierarchy $\widetilde{L}$. Similarly, we write "$a \in C$ with type α" to also specify the declared type α. Similar notations also hold for method names m.

Table 1 inductively defines the partial function $type(C)$ on the class hierarchy $\widetilde{L}$ (of course this induction is well-founded since $\widetilde{L}$ is finite); when defined, it returns a record type. In particular, the type of a method is a boolean combination of arrow types declared in the current and in the parent classes. This follows the same line as [13] in order to deal with habitability of types. The condition $\rho(m) \leq \rho'(m)$ imposed in the method declaration is mandatory to assure that the type of C is a subtype of the type of D; without such a condition, it would be possible to have a class whose type is not a subtype of the parent class. If it were the case, type soundness would fail, as the following example shows.

```
class C extends Object {                class D extends C {
    ...                                     ...
    real m(real x) {return x}               compl m(int x) {return x × i}
    real F() {return this.m(3)}             real G() {return this.F()}
}                                       }
```

As usual **int** $\leq$ **real** $\leq$ **compl**. At run time, the function G returns a **complex** number, instead of a **real**. The example shows that, when the method m is overloaded, we have to be sure that the return type is a subtype of the original type. Otherwise, due to the dynamic instantiation of **this**, there may be a type error. A similar argument justifies the condition $\alpha'' \leq \rho'(a)$ imposed for calculating function *type* for field names.

Table 2. Typing Rules

Typing Expressions :

$$(\mathit{subsum})\ \frac{\Gamma \vdash e : \alpha_1 \quad \alpha_1 \leq \alpha_2}{\Gamma \vdash e : \alpha_2} \qquad (\mathit{const})\ \frac{c \in \mathit{Val}_{\mathbb{B}}}{\Gamma \vdash c : \mathbb{B}}$$

$$(\mathit{var})\ \frac{\Gamma(x) = \alpha}{\Gamma \vdash x : \alpha} \qquad (\mathit{field})\ \frac{\Gamma \vdash e : [a : \alpha]}{\Gamma \vdash e.a : \alpha}$$

$$(\mathit{m\text{-}inv})\ \frac{\Gamma \vdash e_2 : [m : \alpha_1 \to \alpha_2] \quad \Gamma \vdash e_1 : \alpha_1}{\Gamma \vdash e_2.m(e_1) : \alpha_2} \qquad (\mathit{rnd})\ \frac{}{\Gamma \vdash \mathbf{rnd}(\alpha) : \alpha}$$

$$(\mathit{new})\ \frac{\begin{array}{c}\mathit{type}(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}] \qquad \Gamma \vdash \widetilde{e} : \widetilde{\beta} \qquad \widetilde{\beta} \leq \widetilde{\alpha} \\ \rho = [\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_i \neg[a'_i : \alpha'_i] \wedge \bigwedge_j \neg[m'_j : \mu'_j] \qquad \rho \not\simeq \mathbf{0}\end{array}}{\Gamma \vdash \mathbf{new}\ C(\widetilde{e}) : \rho}$$

Typing Method Declarations :

$$(\mathit{m\text{-}decl})\ \frac{x : \alpha_1, \mathbf{this} : \mathit{type}(C) \vdash e : \alpha_2}{\vdash_C \alpha_2\ m\ (\alpha_1\ x)\{\mathbf{return}\ \ e\}}$$

Typing Class Declarations :

$$(\mathit{class})\ \frac{\mathit{type}(D) = [\widetilde{b : \beta}, \widetilde{m : \mu}] \quad K = C(\widetilde{\beta\ b}; \widetilde{\alpha\ a})\{\mathbf{super}(\widetilde{b}); \widetilde{\mathbf{this}.a = a}\} \quad \vdash_C \widetilde{M}}{\vdash \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\widetilde{\alpha\ a}\ K\ \widetilde{M}\}}$$

Typing Programs :

$$(\mathit{prog})\ \frac{\vdash \widetilde{L} \qquad \vdash e : \alpha}{\vdash (\widetilde{L}, e)}$$

Let us now consider the typing rules given in Table 2. We assume Γ to be a typing environment, i.e., a finite sequence of α-type assignments to variables. Most rules are very intuitive. Rule (*subsum*) permits to derive for an expression e of type α_1 also a type α_2, if α_1 is a subtype of α_2. Notice that, for the moment, the subtyping relation used in this rule is the one induced by the bootstrap model. In rule (*const*), we assume that, for any basic type $\mathbb{B}$, there exists a fixed set of constants $\mathit{Val}_{\mathbb{B}} \subseteq \mathcal{K}$ such that the elements of this set have type $\mathbb{B}$. Notice that, for any two basic types $\mathbb{B}_1$ and $\mathbb{B}_2$, the sets $\mathit{Val}_{\mathbb{B}_i}$ may have a non empty intersection. Rule (*var*) derives that x has type α, if x is assigned type α in Γ. Let us now concentrate on rules (*field*) and (*m-inv*). Notice that in these two rules the record types are singletons, as it is enough that inside the record type there is just the field or the method that we want to access or invoke. If the record type is more specific (having other fields or methods), we can still get the singleton record by using the subsumption rule. The rules *m-inv* models methods as invariant in their arguments. This is not restrictive, as we can always use subsumption to promote the type of the argument to match the declared type of the method. For rule (*new*), an object creation can be typed by recording the actual type of the arguments passed to the constructor, since we are confining ourselves to the functional fragment of the language. Moreover, like in [13], we can extend the type of the object, by adding any record type that cannot be assigned to it - as long as this does not lead to a contradiction, i.e. a type semantically equivalent to $\mathbf{0}$. This possibility of adding negative record types is not really necessary for programming purposes: it is only needed to ensure that every non-zero type has at least one value of that type. This property guarantees that the interpretation of types as sets of values induces the same subtyping relation as the bootstrap model. Rule (*rnd*) states that $\mathbf{rnd}(\alpha)$ is of type α. Finally, rule (*m-decl*) checks when a method declaration is acceptable for a class C; this can only happen if *type*(C) is defined. Rules (*class*) and (*prog*) check when a class declaration and a program are well-typed and are similar to the ones in FJ.

3.3 Closing the Circle

To close the circle, one should now interpret types as sets of values. In our calculus, a natural notion of value includes the constants and the objects initialized by only passing values to their constructor.

However, as the classes in $\widetilde{L}$ are finite, with these values we are able to inhabit just a finite number of record types. Also, since we have not higher-order values, the μ-types would not be inhabited. This is a major technical difference w.r.t. [13].

To overcome this problem we use the more general notion of *pseudo-value.* A pseudo-value is a closed, well-typed expression that cannot reduce further. The interpretation of an α-type is the set of pseudo-values of that type For μ-types, we interpret an arrow type as a set of pairs (α, w) such that it is possible to assign to the normal form w the return type of μ whenever the input argument of the method is assigned input type of μ i.e., type α, which "closes" the normal form w. The details can be found in [11].

Using the above intuitions, we define the interpretation function $[\![\cdot]\!]_{\mathcal{V}}$ and, consequently, the subtyping relation $\leq_{\mathcal{V}}$. A priori, the new relation $\leq_{\mathcal{V}}$ could be different from $\leq_{\mathcal{B}}$. However, since the definitions of the model, of the language and of the typing rules have been carefully chosen, the two subtyping relations coincide. Hence, we can prove the following result, the proof of which can be found in [11].

Theorem 1. *The bootstrap model* $[\![\cdot]\!]_{\mathcal{B}}$ *induces the same subtyping relation as* $[\![\cdot]\!]_{\mathcal{V}}$.

Table 3. Operational semantics

$$\textit{(f-ax)}\ \frac{\mathit{type}(C) = [\widetilde{a:\alpha}, \widetilde{m:\mu}]}{(\mathbf{new}\ C(\widetilde{u})).a_i \to u_i} \qquad \textit{(f-red)}\ \frac{e \to e'}{e.a \to e'.a}$$

$$\textit{(r-ax)}\ \frac{\vdash e : \alpha}{\mathbf{rnd}(\alpha) \to e} \qquad \textit{(m-ax)}\ \frac{\mathit{body}(m,u,C) = \lambda x.e}{(\mathbf{new}\ C(\widetilde{u'})).m(u) \to e[{}^{u}/_{x}, {}^{\mathbf{new}\ C(\widetilde{u'})}/_{\mathbf{this}}]}$$

$$\textit{(m-red}_1\textit{)}\ \frac{e' \to e''}{e'.m(e) \to e''.m(e)} \qquad \textit{(m-red}_2\textit{)}\ \frac{e' \to e''}{e.m(e') \to e.m(e'')}$$

$$\textit{(n-red)}\ \frac{e_i \to e_i'}{\mathbf{new}\ C(e_1,\ldots,e_i,\ldots,e_k) \to \mathbf{new}\ C(e_1,\ldots,e_i',\ldots,e_k)}$$

4 Operational Semantics and Soundness of the Type System

The operational semantics is defined through the transition rules of Table 3; these are essentially the same as in FJ. There are only two notable differences: we use function *type* to extract the fields of an object, instead of defining an ad-hoc function; function *body* also depends on the (type of the) method argument, necessary for finding the appropriate declaration when we have multimethods.

We fix the set of class declarations $\widetilde{L}$ and define the operational semantics as a binary relation on the expressions of the calculus $e \to e'$, called *reduction relation.* The axiom for field access *(f-ax)* states that, if we try to access the i-th field of an object, we just return the i-th argument passed to the constructor of that object. We have used the premise $\mathit{type}(C) = [\widetilde{a:\alpha}, \widetilde{m:\mu}]$ as we want all the fields of the object instantiating class C: function $\mathit{type}(C)$ provides them in the right order (i.e., the order in which the constructor of class C expects them to be). The axiom for method invocation *(m-ax)* tries to match the argument of a method in the current class and, if a proper type match is not found, it looks up in the hierarchy; these tasks are carried out by function *body*, whose definition is in the following and the if cases are to be considered in order:

$$\mathit{body}(m,u,C) = \begin{cases} \lambda x.e & \text{if } C \text{ contains } \beta\ m(\alpha\ x)\{\mathbf{return}\ e\} \text{ and } \vdash u:\alpha, \\ \mathit{body}(m,u,D) & \text{if } C \text{ extends } D \text{ in } \widetilde{L}, \\ \mathit{UNDEF} & \text{otherwise.} \end{cases}$$

Notice that method resolution is performed at runtime, by keeping into account the *dynamic* type of the argument; this is called *multimethods* and is different from what happens in Java, where method resolution is performed at compile time by keeping into account the *static* type of the argument. A more traditional modeling of overloading is possible and easy to model.

Soundness of the Type System. Theorem 1 does not automatically imply that the definitions put forward in Sections 3 and 4 are "valid" in any formal sense, only that they are mutually coherent. To complete the theoretical treatment, we need to check type soundness, stated by the following theorems. The full proofs can be found in [11].

Theorem 2 (Subject reduction). *If* $\vdash e : \alpha$ *and* $e \to e'$*, then* $\vdash e' : \alpha'$ *where* $\alpha' \leq \alpha$.

Proof. The proof is by induction on the length of $e \to e'$.

Theorem 3 (Progress). *If* $\vdash e : \alpha$ *where* e *is a closed expression, then* e *is a value or there exists* e' *such that* $e \to e'$.

Proof. The proof is by induction on the structure of e.

5 Discussion on the Calculus

5.1 Recursive Class Definitions

It is possible to write recursive class definitions by assuming a special basic value **null** and a corresponding basic type **unit**, having **null** as its only value. In Java, it is assumed that **null** belongs to every class type; here, because of the complex types we are working with (mainly, because of negations), this assumption cannot be done. This, however, enables us to specify when a field can/cannot be **null**; this is similar to what happens in database systems. In particular, lists of integers can now be defined as:

$$
\begin{array}{l}
L_{intList} = \textbf{class } \textit{intList} \textbf{ extends } \textit{Object}\ \{ \\
\qquad \textbf{int } \textit{val}; \\
\qquad (\alpha \vee \textbf{unit})\ \textit{succ}; \\
\qquad \textit{intList}\ (\textbf{int } x, (\alpha \vee \textbf{unit})\ y)\{\textbf{this}.\textit{val} = x; \textbf{this}.\textit{succ} = y\} \\
\qquad \ldots \\
\}
\end{array}
$$

α being the solution of the recursive equation $\alpha = [\textit{val} : \textbf{int}, \textit{succ} : (\alpha \vee \textbf{unit})]$. Now, we can create the list $\langle 1, 2\rangle$ by writing $\textbf{new}\ \textit{intList}(1, \textbf{new}\ \textit{intList}(2, \textbf{null}))$.

5.2 Implementing Standard Multimethods

Usually in object oriented languages, multimethods can be defined within a single class. For simplicity, we have defined a language where at most one definition can be given for a method name in a class.

It is however possible to partially encode multimethods by adding one auxiliary subclass for every method definition. For instance, suppose we want to define twice a multimethod m within class A:

$$
\begin{array}{l}
\textbf{class } A \textbf{ extends } \textit{Object}\ \{ \\
\qquad \ldots \\
\qquad \alpha_1\ m\ (\beta_1\ x)\{\textbf{return } e_1\} \\
\qquad \alpha_2\ m\ (\beta_2\ x)\{\textbf{return } e_2\} \\
\}
\end{array}
$$

We then replace it with the following declarations:

```
class A1 extends Object {          class A extends A1 {
    ...                                ...
    $\alpha_1$ m ($\beta_1$ x){return $e_1$}          $\alpha_2$ m ($\beta_2$ x){return $e_2$}
}                                  }
```

Introducing subclasses is something that must be done with care. Indeed, it is not guaranteed, in general, that the restrictions for the definition of function *type* (see Table 1) are always satisfied. So, in principle, the encoding described above could turn a class hierarchy where the function *type* is well-defined into a hierarchy where it is not. However, this situation never arises if different bodies of a multimethod are defined for inputs of mutually disjoint types, as we normally do. Also, it is not difficult to add to the language a *typecase* construct, similar to the one of ℂDuce, that would allow more expressivity. We did not pursue this approach in the present paper to simplify the presentation.

5.3 Implementing Typical Java-Like Constructs

We now briefly show how we can implement in our framework traditional programming constructs like *if-then-else* and (a structural form of) *instanceof*. Other constructs, like exceptions, sequential composition and loops, can also be defined.

The expression **if** e **then** e_1 **else** e_2 can be implemented by adding to the program the class definition:

```
class Test extends Object {
    $\alpha$ m ({true} x){return $e_1$}
    $\alpha$ m ({false} x){return $e_2$}
}
```

where {**true**} and {**false**} are the singleton types containing only values **true** and **false**, respectively, and α is the type of e_1 and e_2. Then, **if** e **then** e_1 **else** e_2 can be simulated by (**new** *Test*()).$m(e)$. Notice that this term typechecks, since *test* has type $[m : (\{\mathbf{true}\} \to \alpha) \wedge (\{\mathbf{false}\} \to \alpha)] \simeq [m : (\{\mathbf{true}\} \vee \{\mathbf{false}\}) \to \alpha] \simeq [m : \mathbf{bool} \to \alpha]$. Indeed, in [13] it is proved that $(\alpha_1 \to \alpha) \wedge (\alpha_2 \to \alpha) \simeq (\alpha_1 \vee \alpha_2) \to \alpha$ and, trivially, $\{\mathbf{true}\} \vee \{\mathbf{false}\} \simeq \mathbf{bool}$.

The construct e **instanceof** α checks whether e is typeable at α and can be implemented in a way similar to the *if-then-else*:

```
class InstOf extends Object {
    bool $m_{\alpha_1}$($\alpha_1$ x){return true}
    bool $m_{\alpha_1}$($\neg\alpha_1$ x){return false}
    ...
    bool $m_{\alpha_k}$($\alpha_k$ x){return true}
    bool $m_{\alpha_k}$($\neg\alpha_k$ x){return false}
}
```

where $\alpha_1, \ldots, \alpha_k$ are the types occurring as arguments of an **instanceof** in the program. Then, e **instanceof** α can be simulated by (**new** *InstOf*()).$m_\alpha(e)$.

5.4 Nominal Subtyping vs. Structural Subtyping

The semantic subtyping is a way to allow programmers use powerful typing disciplines, but we do not want to bother them with the task of explicitly writing structural types. Thus, we can introduce aliases. We could write

$$
\begin{array}{lll}
L'_{intList} & = & \mathbf{class}\ \mathit{intList}\ \mathbf{extends}\ \mathit{Object}\ \{ \\
& & \quad \mathbf{int}\ \mathit{val}; \\
& & \quad (\mathit{intList} \vee \mathbf{unit})\ \mathit{succ}; \\
& & \quad \mathit{intList}\ (\mathbf{int}\ x, (\mathit{intList} \vee \mathbf{unit})\ y)\{\mathbf{this}.\mathit{val} = x; \mathbf{this}.\mathit{succ} = y\} \\
& & \quad \ldots \\
& & \}
\end{array}
$$

instead of $L_{intList}$ in Section 5.1. Any sequence of class declarations written in this extended syntax can be then compiled into the standard syntax in two steps:

– First, extract from the sequence of class declarations a system of (mutually recursive) type declarations; in doing this, every class name should be considered as a type identifier. Then, solve such a system of equations.
– Second, replace every occurrence of every class name occurring in a type position (i.e., not in a class header nor as the name of a constructor) with the corresponding solution of the system.

For example, the system of equations (actually, made up of only one equation) associated with $L'_{intList}$ is $\mathit{intList} = [\mathit{val} : \mathbf{int}, \mathit{succ} : (\mathit{intList} \vee \mathbf{unit})]$; if we assume that α denotes the solution of such an equation, the class declaration resulting at the end of the compilation is exactly $L_{intList}$ in Section 5.1.

But nominal types can be more powerful than just shorthands. When using structural subtyping, we can interchangeably use two different classes having the very same structure but different names. However, there can be programming scenarios where also the name of the class (and not only its structure) could be needed. A typical example is the use of exceptions, where one usually extends class *Exception* without changing its structure. In such cases, nominal subtyping can be used to enforce a stricter discipline.

We can integrate this form of nominal subtyping in our semantic framework. To do that, we add to each class a hidden field that represents all the nominal hierarchy that can be generated by that class. If we want to be nominal, we will consider also this hidden field while checking subtyping. In practice, the (semantic) ‘nominal’ type of a class is the set of qualified names of all its subclasses; this will enable us to say that C is a ‘nominal’ subtype of D if and only if C’s subclasses form a subset of D’s ones. Notice that working with subsets is the key feature of our semantic approach to subtyping. This is the reason why we need types as sets and, e.g., cannot simply add to objects a field with the class they are instance of.

It remains to describe how we can use nominal subtyping in place of the structural one. We propose two ways. In declaring a class or a field, or in the return type of a method, we could add the keyword **nominal**, to indicate to the compiler that nominal subtyping should always be used with it. However, the only place where subtyping is used is in function *body*, i.e. when deciding which body of an overloaded method we have to activate on a given sequence of actual values. Therefore, we could be even more flexible, and use the keyword **nominal** in method declarations, to specify which method arguments have to be checked nominally and which ones structurally. For example, consider the following class declaration:

$$\begin{array}{l}\textbf{class}\ A\ \textbf{extends}\ \mathit{Object}\ \{ \quad \ldots \\ \quad \textbf{int}\ m\ (C\ x,\ \textbf{nominal}\ C\ y)\{\ \textbf{return}\ 0;\ \} \\ \}\end{array}$$

Here, every invocation of method m will check the type of the first argument structurally and the type of the second one nominally. Thus, if we consider the following class declarations

$$\textbf{class}\ C\ \textbf{extends}\ \mathit{Object}; \quad \{\ \}\ \textbf{class}\ D\ \textbf{extends}\ \mathit{Object}\ \{\ \}$$

the expressions $(\textbf{new}\ A()).m(\textbf{new}\ C(), \textbf{new}\ C())$, $(\textbf{new}\ A()).m(\textbf{new}\ D(),$ $\textbf{new}\ C())$ and $(\textbf{new}\ A()).m(\textbf{new}\ \mathit{Object}(), \textbf{new}\ C())$ typecheck, whereas $(\textbf{new}\ A()).m(\textbf{new}\ C(), \textbf{new}\ D())$ and $(\textbf{new}\ A()).m(\textbf{new}\ C(), \textbf{new}\ \mathit{Object}())$ do not.

6 Conclusions and Future Work

We have presented a Java-like programming framework that integrates structural subtyping, boolean connectives and semantic subtyping to exploit and combine the benefits of such approaches. There is still work to do in this research line.

This paper lays out the foundations for a concrete implementation of our framework. First of all, a concrete implementation calls for algorithms to decide the subtyping relation; then, decidability of subtyping is exploited to define a typechecking algorithm for our type system. This can be achieved by adding algorithms similar to those in [13]. A preliminary formal development can be found in the first author's M.S thesis [11]. These are intermediate steps towards a prototype programming environment where writing and evaluating the performances of code written in the new formalism.

Another direction for future research is the enhancement of the language considered. For example, one can consider the extension of FJ with assignments; this is an important aspect because mutable values are crucial for modeling the heap, a key feature in object oriented programming. We think that having a state would complicate the issue of typing, because of the difference between the declared and the actual type of an object. Some ideas on how to implement the mutable state can come from the choice made in the implementation of $\mathbb{C}$Duce. The fact that we have assumed nondeterministic methods can also help in modeling a mutable state: as we have said, the input-output behavior of a function can be seen as nondeterministic since, besides its input, the function has access to the state.

Another possibility for enhancing the language is the introduction of higher-order values, in the same vein as the Scala programming language [21]; since the framework of [13] is designed for a higher-order language, the theoretical machinery developed therein should be easily adapted to the new formalism.

References

1. Abadi, M., Cardelli, L.: A Theory of Primitive Objects - Untyped and First-Order Systems. In: Hagiya, M., Mitchell, J.C. (eds.) TACS 1994. LNCS, vol. 789, pp. 296–320. Springer, Heidelberg (1994)
2. Agrawal, R., de Michiel, L.G., Lindsay, B.G.: Static type checking of multimethods. In: Proc. of OOPSLA, pp. 113–128. ACM Press (1991)
3. Aiken, A., Wimmers, E.L.: Type inclusion constraints and type inference. In: Proc. of FPCA, pp. 31–41. ACM (1993)
4. Ancona, D., Lagorio, G.: Coinductive type systems for object-oriented languages. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 2–26. Springer, Heidelberg (2009)
5. Boyland, J., Castagna, G.: Type-safe compilation of covariant specialization: A practical case. In: Cointe, P. (ed.) ECOOP 1996. LNCS, vol. 1098, pp. 3–25. Springer, Heidelberg (1996)
6. Boyland, J.T., Castagna, G.: Parasitic Methods: an implementation of multi-methods for Java. In: Proc. of OOPSLA. ACM Press (1997)
7. Castagna, G.: Object-oriented programming: a unified foundation. Progress in Theoretical Computer Science series. Birkäuser, Boston (1997)
8. Castagna, G.: Semantic subtyping: Challenges, perspectives, and open problems. In: Coppo, M., Lodi, E., Pinna, G.M. (eds.) ICTCS 2005. LNCS, vol. 3701, pp. 1–20. Springer, Heidelberg (2005)
9. Castagna, G., De Nicola, R., Varacca, D.: Semantic subtyping for the pi-calculus. Theoretical Computer Science 398(1-3), 217–242 (2008)
10. Damm, F.M.: Subtyping with union types, intersection types and recursive types. In: Hagiya, M., Mitchell, J.C. (eds.) TACS 1994. LNCS, vol. 789, pp. 687–706. Springer, Heidelberg (1994)
11. Dardha, O.: Sottotipaggio semantico per linguaggi ad oggetti. MS thesis, Dip. Informatica, "Sapienza" Univ. di Roma, `http://www.dsi.uniroma1.it/~gorla/TesiDardha.pdf`
12. Findler, R.B., Flatt, M., Felleisen, M.: Semantic casts: Contracts and structural subtyping in a nominal world. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 365–389. Springer, Heidelberg (2004)
13. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. Journal of the ACM 55(4), 1–64 (2008)
14. Gil, J., Maman, I.: Whiteoak: introducing structural typing into Java. In: Proc. of OOPSLA, pp. 73–90. ACM (2008)
15. Hosoya, H., Pierce, B.C.: Regular expression pattern matching for XML. SIGPLAN Notices 36(3), 67–80 (2001)
16. Hosoya, H., Pierce, B.C.: Xduce: A statically typed XML processing language. ACM Transactions on Internet Technology 3(2), 117–148 (2003)
17. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems 23(3), 396–450 (2001)

18. Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J.: The Objective Caml system, release 3.11 (2008)
19. Malayeri, D., Aldrich, J.: Integrating nominal and structural subtyping. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 260–284. Springer, Heidelberg (2008)
20. Malayeri, D., Aldrich, J.: Is structural subtyping useful? An empirical study. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 95–111. Springer, Heidelberg (2009)
21. Odersky, M., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the Scala programming language. Technical report (2004)
22. Ostermann, K.: Nominal and structural subtyping in component-based programming. Journal of Object Technology 7(1), 121–145 (2008)
23. Pearce, D.J.: Sound and complete flow typing with unions, intersections and negations. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 335–354. Springer, Heidelberg (2013)
24. Rémy, D., Vouillon, J.: Objective ML: A simple object-oriented extension of ML. In: Proc. of POPL, pp. 40–53. ACM (1997)
25. Sangiorgi, D., Walker, D.: The Pi-Calculus: A Theory of Mobile Processes. Cambridge University Press, New York (2003)

Polymorphic Types for Leak Detection in a Session-Oriented Functional Language

Viviana Bono, Luca Padovani, and Andrea Tosatto

Dipartimento di Informatica, Università di Torino, Italy

Abstract. Copyless message passing is a communication paradigm in which only pointers are exchanged between sender and receiver processes. Because of its nature, this paradigm requires that messages are treated as *linear* resources. Yet, even linear type systems leave room for scenarios where apparently well-typed programs may leak memory. In this work we develop a polymorphic type system for leak-free copyless messaging in a functional setting, where first-class functions can be used as messages.

1 Introduction

When communicating processes can access a shared address space, it is sensible to consider a *copyless* form of communication whereby only pointers to messages (instead of the messages themselves) are copied from senders to receivers. The Singularity Operating System [9,10] is a notable example of system making pervasive use of copyless communication. In Singularity, messages live in a shared area called *exchange heap* that, for practical reasons, cannot be garbage collected: data in this area must be explicitly allocated and deallocated. Messages travel through channels that are represented as pairs of *peer endpoints*: a message sent over one endpoint is received from the corresponding peer. Because channel endpoints can be sent as messages, they are also allocated in the exchange heap and explicitly managed.

Explicit memory management is a well-known source of hard-to-trace bugs. For this reason, it calls for the development of static analysis techniques meant to spot dangerous code. In [1,2] we have developed a type system for a language of processes that interact through copyless messaging: well-typed processes are guaranteed to be free from memory faults, memory leaks, and communication errors. The type system associates channel endpoints with *endpoint types* reminiscent of session types [7,8]. The present work extends the results of [2] to a language with first-class functions. For example,

$$g \overset{\text{def}}{=} \lambda c.\lambda x.\texttt{let}\ f, c' = \texttt{receive}\ c\ \texttt{in}\ \texttt{close}\ c'; (f\ x)$$

is a function that, when applied to a channel endpoint c and a value x, transforms x through a function received from c. The $\texttt{receive}\ c$ application evaluates to a pair consisting of the message received from c and c itself, the $\texttt{let}$ deconstructs such pair and binds its components to the local variables f and c', and `close` c' deallocates c'. The explicit re-binding of c enables the type system to keep track of resource allocation and to spot violations in the memory management. Indeed, endpoints are linear resources that are *consumed* when used in a function application (like c and c' in `receive` c and `close` c') and *acquired* when obtained as result of a function application (like for f and c' returned by `receive`). In addition, the re-binding allows to assign different types to the same channel endpoint according to how the code uses it: the above function can be typed with the assignments $c : \texttt{?}(\texttt{Int} \multimap \texttt{Int}).\texttt{end}$ and $c' : \texttt{end}$ where the type of c denotes the fact that it can be used for receiving a message of type $\texttt{Int} \multimap \texttt{Int}$ (a linear function from integers to integers) and the type `end` of c' is the residual of c's type after receiving this message; `end` indicates that c' can be deallocated.

In [2] it was observed that it is possible to write apparently correct code that yields *memory leaks*, whereby an allocated region of the heap becomes inaccessible. This phenomenon manifests itself in the program

$$\texttt{let}\ a, b = \texttt{open}\ \texttt{unit}\ \texttt{in}\ \texttt{close}\ (\texttt{send}\ b\ a)$$

which creates a new channel represented as the two peer endpoints a and b and sends b over its own peer a. This code fragment can be typed using the assignment $\{a : T_1, b : S_1\}$ where $T_1 = \texttt{!}S_1.\texttt{end}$ and S_1 is the recursive type satisfying the equation $S_1 = \texttt{?}S_1.\texttt{end}$. Note that in this code fragment every resource that is acquired is also consumed. Yet, after the execution of this code only endpoint a is actually deallocated, while endpoint b has become inaccessible because stored within its own queue. In [2] we rule out code like this by restricting the values that can be sent as messages depending on their type. The idea consists in looking at types for estimating the length of the chains of pointers originating from values with that type – we call such measure *type weight* – and then restricting messages to values whose type has a bounded weight. For example, the queue of an endpoint of type S_1 may contain a message of type S_1, therefore the weight of S_1 is unbounded, whereas the weight of T_1 is zero because a can only be used for sending messages, so its queue will never contain a message.

It turns out that the same technique does not work "out of the box" in a language with first-class functions. The problem is that arrow types only tell us what a function accepts and produces, but not which other (heap-allocated) values the function may use, while this information is essential for determining the weight of an arrow type. To illustrate the issue, consider the code fragment

$$\texttt{let}\ a, b = \texttt{open}\ \texttt{unit}\ \texttt{in}\ \texttt{close}\ (\texttt{send}\ (g\ b)\ a)$$

which is a little twist from the previous one. According to the definition of g, $(g\ b)$ is a message that *contains* b. This code fragment can be typed with the assignment $\{a : T_2, b : S_2\}$, where $T_2 = \texttt{!}(\texttt{Int} \multimap \texttt{Int}).\texttt{end}$ and $S_2 = \texttt{?}(\texttt{Int} \multimap \texttt{Int}).\texttt{end}$. As before, this closed code fragment yields a memory leak due to b not being deallocated, but in this case the type $\texttt{Int} \multimap \texttt{Int}$ of the message in S_2 does not provide much information: we only know that $(g\ b)$ is a function that may make use of a linear value, but the type of such linear value, which is key in order to assess the weight of $\texttt{Int} \multimap \texttt{Int}$, is unknown. The solution we put forward consists in decorating linear arrow types with an explicit weight, as in $\texttt{Int}\,w\!\multimap\texttt{Int}$, to keep track of this information. In the above example, the weight of S_2 should be strictly greater than w, because endpoint b carries messages of type $\texttt{Int}\,w\!\multimap\texttt{Int}$. At the same time, w should not be smaller than the weight of S_2, because $(g\ b)$ contains b. From this train of thoughts, one infers that there is no finite bound for w, and consequently that $(g\ b)$ cannot be safely sent over a.

Polymorphism adds another dimension to the problem and forces us to consider a more structured representation of type weights. For example, the function

$$\mathit{forward} \stackrel{\text{def}}{=} \lambda x.\lambda y.\texttt{let}\ m, x' = \texttt{receive}\ x\ \texttt{in}\ (x', \texttt{send}\ m\ y)$$

which forwards a message from an endpoint x to another endpoint y, can be given the polymorphic type $\texttt{?}a.A \to \texttt{!}a.B\ w\!\multimap A \otimes B$. The issue is how to determine the weight w, given that x occurs free in the function $\lambda y \cdots$ and that it has the partially specified type $\texttt{?}a.A$. The actual weight of $\texttt{?}a.A$ *depends* on the weight of the types with which a and A are instantiated. In particular, it is the maximum between the weight of A and the weight of a plus 1 (because the queue for x may contain a value of type a). We keep track of this dependency by letting $w = \{a, A\} + 1$.

In the rest of the paper we formalize all the notions sketched so far. We begin by defining syntax and reduction semantics of a core functional language equipped with session-oriented communication primitives (Section 2). We also provide a precise definition of "correct" programs as those that are free from memory faults, memory leaks, and communication errors. We proceed by presenting the type language (Section 3), the type system and its soundness results (Section 4). Related work (Section 5) and a few concluding remarks (Section 6) end the main body of the paper. Proofs of the results can be found in the long version of the paper [3].

2 Language

The syntax of our language is described in Table 1, where we use the following syntactic categories: x, y range over an infinite set of *variables*; p, q range over an infinite set $\mathsf{Pointers}$ of *pointers*; u ranges over *names*, which are either variables or pointers, and $\mathtt{U}$, $\mathtt{V}$ over sets of names; E ranges over *expressions* and v over *values*; we write $\tilde{v}$ to denote *queues*, namely finite sequences of values; $\texttt{k}$ ranges over *constants* from the set $\{\texttt{unit}, \texttt{fix}, \texttt{fork}, \texttt{open}, \texttt{close}, \texttt{send}, \texttt{receive}\}$; P, Q range over *processes*; μ ranges over *heaps*. The sub-language of expressions is almost standard, except for the $\texttt{let}$ construct which deconstructs pairs and binds their components to two variables. As usual, $\lambda x.E$ binds x in E and $\texttt{let}\ x, y = E_1\ \texttt{in}\ E_2$ binds x and y in E_2, therefore, bound and free names are defined in the usual way. We will sometimes write $\texttt{let}\ x = E_1\ \texttt{in}\ E_2$ in place of $\texttt{let}\ x, y = (E_1, \texttt{unit})\ \texttt{in}\ E_2$ where y is some fresh variable. Processes are parallel compositions of expressions, each expression representing a thread of execution. We identify processes modulo commutativity and associativity of $\|$.

Table 1. Syntax of expressions, processes, values, and heaps

$E ::=$		**Expression**	$v ::=$		**Value**
	v	(value)		$\mathtt{p}$	(pointer)
$\mid$	x	(variable)	$\mid$	$\mathtt{k}$	(constant)
$\mid$	(E, E)	(pair)	$\mid$	$\lambda x.E$	(abstraction)
$\mid$	EE	(application)	$\mid$	(v, v)	(pair)
$\mid$	$\mathtt{let}\ x, y = E\ \mathtt{in}\ E$	(pattern match)			
			$\mu ::=$		**Heap**
$P ::=$		**Process**		$\varnothing$	(empty heap)
	$\langle E \rangle$	(thread)	$\mid$	$\mathtt{p} \mapsto [\mathtt{q}; \tilde{v}]$	(endpoint)
$\mid$	$P \parallel P$	(composition)	$\mid$	μ, μ	(composition)

We write $\mathsf{fn}(E)$ and $\mathsf{fn}(P)$ for denoting the set of free names occurring in E and P.

In order to express the operational semantics of processes, we need an explicit representation of *heaps* as finite maps from pointers to endpoint structures $[\mathtt{p}, \tilde{v}]$, which, in turn, are a pair containing a pointer and a queue of values, representing the messages received at that endpoint. The set of pointers to allocated endpoint structures is $\mathsf{dom}(\mu)$, and we assume that the composition μ_1, μ_2 is defined only when $\mathsf{dom}(\mu_1) \cap \mathsf{dom}(\mu_2) = \emptyset$. A *system* is a pair $\mu \mathbin{;} P$ of a heap μ and a process P.

Table 2 defines the reduction semantics of expressions and of systems. Expressions reduce according to a conventional call-by-value semantics extended with pattern matching over pairs. Systems reduce as a consequence of expressions that are evaluated in threads and of primitive functions forking new threads and implementing the communication operations. Rule (R-Thread) performs a step of computation within a thread. The *evaluation context* $\mathcal{E}$ [12,6] is an expression with a hole, denoted by $[\ ]$, where computation in a thread happens next. Evaluation contexts are defined by

$$\mathcal{E} ::= [\ ] \mid (\mathcal{E}, E) \mid (v, \mathcal{E}) \mid \mathcal{E}E \mid v\mathcal{E} \mid \mathtt{let}\ x, y = \mathcal{E}\ \mathtt{in}\ E \mid \mathtt{let}\ x, y = v\ \mathtt{in}\ \mathcal{E}$$

and $\mathcal{E}[E]$ denotes the result of filling the hole in $\mathcal{E}$ with the expression E.

Rule (R-Par) singles out threads running in parallel. Rule (R-Fork) spawns a new thread. Rule (R-Open) creates a channel as a pair of peer endpoints, by allocating two endpoint structures in the heap which point to each other and initially have an empty queue (ε denotes the empty sequence of values). Rule (R-Send) inserts a value v on the queue of the peer endpoint of $\mathtt{p}$. Rule (R-Receive) extracts the head value from the queue associated with the endpoint pointed to by $\mathtt{p}$. In both `send` and `receive`, the operation evaluates to the endpoint being used for communication, which is thus available for further operations. In the following we write $\Longrightarrow$ for the reflexive, transitive closure of $\longrightarrow$ and we write $\mu \mathbin{;} P \longrightarrow\!\!\!\!\!\!/\ \ $ if there exist no μ' and P' such that $\mu \mathbin{;} P \longrightarrow \mu' \mathbin{;} P'$.

In this work, as in [2], we focus on three properties of systems: we wish every system to be *fault free*, where a fault is an attempt to use a pointer not

Table 2. Reduction semantics of expressions and systems

Reduction of expressions

$$(\lambda x.E)v \longrightarrow_{\mathsf{v}} E\{v/x\} \qquad \texttt{fix}(\lambda x.E) \longrightarrow_{\mathsf{v}} E\{\texttt{fix}(\lambda x.E)/x\}$$

$$\texttt{let}\ x, y = (v, w)\ \texttt{in}\ E \longrightarrow_{\mathsf{v}} E\{v, w/x, y\}$$

Reduction of systems

(R-Thread)
$$\frac{E \longrightarrow_{\mathsf{v}} E'}{\mu \fatsemi \langle \mathcal{E}[E]\rangle \longrightarrow \mu \fatsemi \langle \mathcal{E}[E']\rangle}$$

(R-Fork)
$$\mu \fatsemi \langle \mathcal{E}[\texttt{fork}\ E]\rangle \longrightarrow \mu \fatsemi \langle \mathcal{E}[\texttt{unit}]\rangle \parallel \langle E\rangle$$

(R-Par)
$$\frac{\mu \fatsemi P_1 \longrightarrow \mu' \fatsemi P_1'}{\mu \fatsemi P_1 \parallel P_2 \longrightarrow \mu' \fatsemi P_1' \parallel P_2}$$

(R-Open)
$$\mu \fatsemi \langle \mathcal{E}[\texttt{open unit}]\rangle \longrightarrow \mu, \mathsf{p} \mapsto [\mathsf{q}; \varepsilon], \mathsf{q} \mapsto [\mathsf{p}; \varepsilon] \fatsemi \langle \mathcal{E}[(\mathsf{p}, \mathsf{q})]\rangle$$

(R-Send)
$$\mu, \mathsf{p} \mapsto [\mathsf{q}; \mathcal{Q}], \mathsf{q} \mapsto [\mathsf{p}; \mathcal{Q}'] \fatsemi \langle \mathcal{E}[\texttt{send}\ (v, \mathsf{p})]\rangle \longrightarrow \mu, \mathsf{p} \mapsto [\mathsf{q}; \mathcal{Q}], \mathsf{q} \mapsto [\mathsf{p}; \mathcal{Q}'v] \fatsemi \langle \mathcal{E}[\mathsf{p}]\rangle$$

(R-Receive)
$$\mu, \mathsf{p} \mapsto [\mathsf{q}; v\mathcal{Q}] \fatsemi \langle \mathcal{E}[\texttt{receive}\ \mathsf{p}]\rangle \longrightarrow \mu, \mathsf{p} \mapsto [\mathsf{q}; \mathcal{Q}] \fatsemi \langle \mathcal{E}[(v, \mathsf{p})]\rangle$$

corresponding to an allocated endpoint; we wish every system to be *leak free*, where a leak is an endpoint that becomes unreachable because no reference to it is directly or indirectly available to the processes in the system; finally, we wish every system to avoid *communication errors*, by enjoying (a limited form of) progress, meaning that no process in the system should get stuck while reading messages from a non-empty queue. We conclude this section by making these properties precise. In order to do so, we need to formalize the reachability of a heap object with respect to a set of *root* pointers. Intuitively, a process P may directly reach any object located at some pointer in the set $\mathsf{fn}(P)$ (we can think of the pointers in $\mathsf{fn}(P)$ as of the local variables of the process stored in its stack); from these pointers, the process may reach transitively other heap objects by reading messages from the queue of the endpoints it can reach.

Definition 2.1 (reachable pointers). *We say that* p *is reachable from* q *in* μ *(written* $\mathsf{p} \prec_\mu \mathsf{q}$*) if* $\mathsf{q} \mapsto [\mathtt{r}; \tilde{w}v\tilde{w}'] \in \mu$ *and* $\mathsf{p} \in \mathsf{fn}(v)$*. We write* $\preceq_\mu$ *for the reflexive and transitive closure of* $\prec_\mu$*. The pointers reachable from* $\mathtt{U}$ *in* μ *are defined as* $\mu\text{-}\mathsf{reach}(\mathtt{U}) = \{\mathsf{p} \in \mathsf{Pointers} \mid \exists \mathsf{q} \in \mathtt{U} : \mathsf{p} \preceq_\mu \mathsf{q}\}$.

We are now ready to define formally well-behaved processes.

Definition 2.2 (well-behaved process). *We say that* P *is* well behaved *if for every possible reduction* $\varnothing \fatsemi P \Rightarrow \mu \fatsemi Q$ *the following properties hold:*

Table 3. Syntax of types

$\sigma ::=$		**Type Scheme**	$T ::=$		**Endpoint Type**
	t	(monomorphic type)		`end`	(termination)
$\mid$	$\forall\alpha :: \rho.\sigma$	(polymorphic type)	$\mid$	A	(variable)
			$\mid$	$\overline{A}$	(dualized variable)
$t ::=$		**Type**	$\mid$	$?t.T$	(input)
	`Unit`	(unit type)	$\mid$	$!t.T$	(output)
$\mid$	a	(variable)	$\mid$	`rec` $A.T$	(recursion)
$\mid$	T	(endpoint type)			
$\mid$	$t \otimes t$	(linear pair)	$w ::=$		**Weight**
$\mid$	$t \rightarrow t$	(function)		∞	(unbounded weight)
$\mid$	$t\, w\!\multimap t$	(linear function)	$\mid$	$X + n$	(bounded weight)

1. $\mathsf{dom}(\mu) = \mu\text{-}\mathsf{reach}(\mathsf{fn}(Q))$.
2. *if* $Q = P_1 \parallel P_2$, *then* $\mu\text{-}\mathsf{reach}(\mathsf{fn}(P_1)) \cap \mu\text{-}\mathsf{reach}(\mathsf{fn}(P_2)) = \emptyset$.
3. *if* $Q = \langle E \rangle \parallel Q'$ *and* $\mu \,⨾\, \langle E \rangle \not\longrightarrow$, *then either* $E =$ `unit`, *or* $E = \mathcal{E}[$`receive p`$]$ *and* $\mathsf{q} \mapsto [\mathsf{p}; \varepsilon] \in \mu$, *or* $E = \mathcal{E}[$`close p`$]$ *and* $\mathsf{p} \mapsto [\mathsf{q}; \varepsilon] \in \mu$.

Conditions (1) and (2) ask for the absence of faults and leaks. In detail, condition (1) states that every allocated pointer in the heap is reachable by one process, and that every reachable pointer corresponds to an object allocated in the heap. Condition (2) states that processes are isolated, namely that no pointer is reachable from two or more processes. Since expressions of the form `close p` are persistent (they do not reduce), this condition rules out memory faults whereby the same endpoint is deallocated multiple times. Condition (3) requires the absence of communication errors, namely that if $\mu \,⨾\, Q$ is stuck (no reduction is possible), then it is because every non-terminated process in Q is waiting for a message on an endpoint having an empty queue. This configuration corresponds to a genuine deadlock where every process in some set is waiting for a message that is to be sent by another process in the same set. Condition (3) also ensures the absence of so-called *orphan messages*: no message accumulates in the queue of closed endpoints.

3 Types

Table 3 gives the syntax of types using the following syntactic categories: m, n range over natural numbers; A, B, ... range over an infinite set of *endpoint type variables*; a, b, ... range over an infinite set of *value type variables*; α, β range over *type variables*, which are either endpoint or value type variables without distinction; X, Y range over finite sets of type variables; ρ ranges over *qualifiers*, which are elements of $\{$`any`$,$`fin`$\}$; w ranges over *weights*; t, s range over *types*; σ range over *type schemes*; T, S range over *endpoint types*.

Endpoint types denote pointers to channel endpoints; they are fairly standard session types with input/output prefixes $?t$/$!t$, recursion, and a terminal state `end`. Endpoint type variables A can occur in *dualized form* $\overline{A}$, as in [4]. This is necessary for typing some functions, beside simplifying the definition of *duality*. For simplicity we omit choices and branches; they can be added without posing substantial problems. *Types* include the conventional constructs of functional languages *à la ML*, comprising a `Unit` type (other data types can be added as needed), linear functions, and linear pairs. The linear types are necessary to denote objects (functions, pairs) that contain channel endpoints and that, for this reason, must be owned and used linearly. In particular, the linear arrow type $t \; w\!\multimap s$ denotes a function whose body may contain pointers and has an explicit decoration w determining its weight. A *weight* is a term representing the length of a chain of pointers in the program heap. It can be either ∞, denoting an unbound length, or $X + n$ denoting a length that is bound by the weight of the types that will instantiate the type variables in X plus the value of the constant n. We will often write X instead of $X + 0$ and n instead of $\emptyset + n$. *Type schemes* are almost standard, except that polymorphic type variables are associated with a *qualifier* ρ: if the qualifier is `any`, then there is no constrain as to which types may instantiate the type variable; if the qualifier is `fin`, then only finite-weight types may instantiate the type variable. We will write $\tilde{t}$ for denoting sequences $t_1, \ldots, t_n$ of types and we will often write $\forall\tilde{\alpha} :: \tilde{\rho}.t$ in place of $\forall\alpha_1 :: \rho_1 \cdots \forall\alpha_n :: \rho_n.t$ for some n.

A type is *well formed* if none of its endpoint type variables bound by a `rec` occurs in a weight. For example, both $\texttt{Unit}\{A\}\!\multimap\texttt{Unit}$ and $\forall A :: \texttt{any}.\texttt{Unit}\{A\}\!\multimap$ `Unit` are well formed, but $\texttt{rec}\ A.!(\texttt{Unit}\ \{A\}\!\multimap \texttt{Unit}).\texttt{end}$ is not. From now on we implicitly assume to work with well-formed types.

The predicate $\mathsf{lin}(\sigma)$ identifies *linear types*:

$$\mathsf{lin}(\alpha) \qquad \mathsf{lin}(T) \qquad \mathsf{lin}(t_1 \otimes t_2) \qquad \mathsf{lin}(t_1 \; w\!\multimap t_2) \qquad \frac{\mathsf{lin}(t)}{\mathsf{lin}(\forall\tilde{\alpha} :: \tilde{\rho}.t)}$$

We say that σ is *unlimited*, notation $\mathsf{un}(\sigma)$, if not $\mathsf{lin}(\sigma)$. Note that a type variable is always considered linear because it *may* be instantiated by a linear type. A full-fledged type system might distinguish between linear and unlimited type variables for better precision; we leave this as a straightforward extension for the sake of simplicity.

There are three crucial notions regarding types that we need to define next, namely *duality*, *type weight*, *substitution*. It turns out that these notions are mutually dependent on one another and their formal definition requires a carefully ordered sequence of intermediate steps that relies on type well formedness. Here we only present the "final" definitions and highlight peculiarities and pitfalls of each, while the detailed development can be found in [3].

Duality. Communication errors are prevented by associating peer endpoints with dual endpoint types, so that when one endpoint type allows sending a message of type t, the dual endpoint type allows receiving messages of type t and when one endpoint should be closed the other endpoint should be closed as well. Roughly, the dual of an endpoint type T, denoted by $\overline{T}$, is obtained from T by swapping

?'s with !'s so that, for example, the dual of $?t.!s.\texttt{end}$ is $!t.?s.\texttt{end}$. In practice, things are a little more complicated because of recursive behaviors. For example, the dual of $T = \texttt{rec}\ A.!A.\texttt{end}$ is *not* $S = \texttt{rec}\ A.?A.\texttt{end}$. Indeed, in T the recursion variable occurs within a prefix, denoting the fact that an endpoint of type T carries messages which have themselves type T. That is, $T = !T.\texttt{end}$. By contrast, we have $S = ?S.\texttt{end}$, hence from an endpoint of type S we can receive another endpoint having type S. In fact, we have $\overline{T} = ?T.\texttt{end} \neq S$.

The *dual* of an endpoint type is inductively defined by the equations:

$$\begin{array}{ccc} \overline{\texttt{end}} = \texttt{end} & \overline{A} = \overline{A} & \overline{?t.T} = !t.\overline{T} \\ \overline{\texttt{rec}\ A.T} = \texttt{rec}\ A.\overline{T\{\overline{A}/A\}} & \overline{\overline{A}} = A & \overline{!t.T} = ?t.\overline{T} \end{array}$$

where $T\{\overline{A}/A\}$ denotes the endpoint type T where free occurrences of A have been replaced by its dualized form and free occurrences of $\overline{A}$ by A. For example, we have $\overline{\texttt{rec}\ A.!A.\texttt{end}} = \texttt{rec}\ A.\overline{!\overline{A}.\texttt{end}} = \texttt{rec}\ A.?\overline{A}.\texttt{end}$.

Weight. The weight of a type (scheme) gives information about the length of the chains of pointers originating from values having that type (scheme). For example, the weight of end is 0, because the queue of an endpoint of type end will never contain any message, hence no chains of pointers can originate from an endpoint of this type. On the contrary, an endpoint of type ?end.end *may* contain a pointer to another endpoint of type end, therefore its weight is 1. Because types may contain type variables, in general the weight of a type depends on how these type variables are instantiated. In order to compute the weight of a type, we must be able to compare weights:

Definition 3.1 (weight order). *We define the relation $\leq$ over weights as the least partial order such that $w \leq \infty$ and $X + m \leq Y + n$ if $X \subseteq Y$ and $m \leq n$.*

Observe that, if $\mathcal{W}$ is the set of all weights, then $(\mathcal{W}, \leq)$ is a complete lattice with least element $\emptyset + 0$ and greatest element ∞. In what follows we will use the operators $\vee$ and $\wedge$ to respectively compute the join and meet of possibly infinite sets of weights.

Definition 3.2 (weight). *Let $\downarrow$ be the largest relation such that $t \downarrow w$ implies either*

- *$w = \infty$, or*
- *$t = \texttt{Unit}$ or $t = t_1 \to t_2$ or $t = \texttt{end}$ or $t = !s.T$, or*
- *$t = \alpha$ and $w = (X \cup \{\alpha\}) + n$, or*
- *$t = t_1 \otimes t_2$ and $t_1 \downarrow w$ and $t_2 \downarrow w$, or*
- *$t = ?s.T$ and $w = X + (n+1)$ and $s \downarrow (X + n)$ and $T \downarrow w$, or*
- *$t = t_1\, w' \!\multimap t_2$ and $w' \leq w$.*

The weight *of a type t, denoted $\|t\|$, is defined as $\|t\| \stackrel{\text{def}}{=} \bigwedge_{t\downarrow w} w$.*

Intuitively, the relation $t \downarrow w$ says that w is an upper bound for the length of the chains of pointers originating from values of type t, and $\|t\|$ is the least of such upperbounds. It is easy to see that every unlimited type has a null weight (a value with unlimited type cannot contain any pointer) and that, for instance, $\|\alpha\| = \{\alpha\}$ and $\|t \otimes s\| = \|t\| \vee \|s\|$. Also, endpoints with type $\texttt{end}$ or $\texttt{!}t.T$ have null weight because their queues must be empty (this property will be enforced by the type system in Section 4). However, we have that $\|\texttt{?}a.\texttt{end}\| = \{a\} + 1$ because an endpoint of such type may contain a value of type a, so the length of the longest chain of pointers originating from such an endpoint is 1 plus the length of longest chain of pointers originating from a value with type that instantiates a. In general, we have $\|\texttt{?}t.T\| = (\|t\| + 1) \vee \|T\|$. If we take the endpoint type $S_1 = \texttt{rec}\ A.\texttt{?}A.\texttt{end}$ from Section 1 we have $\|S_1\| = \infty$ because S_1 has no finite upperbound. Finally, note that $\|\overline{A}\| = \infty$. This is because, in general, there is no relationship between the weight of an endpoint type and that of its dual. For instance, we have $\|\texttt{!}S_1.\texttt{end}\| = 0$ but $\|\overline{\texttt{!}S_1.\texttt{end}}\| = \|\texttt{?}S_1.\texttt{end}\| = \infty$. It would be possible to allow dualized type variables in the syntax of weights, but since such variables occur seldom in types we leave this extension out of our formal treatment and conservatively approximate their weight to ∞.

Substitution. Intuitively, a substitution $t\{s/\alpha\}$ represents the type obtained by replacing the occurrences of α in t with s. This notion is standard, except for two features that are specific of our type language. The first feature is the presence of dualized endpoint type variables $\overline{A}$. The idea is that, when A is replaced by an endpoint type T, $\overline{A}$ is replaced by $\overline{T}$, namely by the dual endpoint type of T that we have just introduced. The second feature is the presence of type variables in weights which decorate linear function types. In particular, a substitution $(t_1\ w\!\multimap t_2)\{s/\alpha\}$ may need to update $w = X + n$ if $\alpha \in X$. Formally, we define a weight substitution operation $w\{w'/\alpha\}$ such that

$$w\{w'/\alpha\} \stackrel{\text{def}}{=} \begin{cases} ((X \setminus \{\alpha\}) \vee w') + n & \text{if } w = X + n \text{ and } \alpha \in X \\ w & \text{otherwise} \end{cases}$$

where we define a meta operator $w + n$ such that $\infty + n = \infty$ and $(X + m) + n = X + (m + n)$. Then $t\{s/\alpha\}$ is defined in the standard way except that

$$\overline{A}\{T/A\} = \overline{T} \qquad \text{and} \qquad (t_1\ w\!\multimap t_2)\{s/\alpha\} = t_1\{s/\alpha\}\ w\{\|s\|/\alpha\}\!\multimap t_2\{s/\alpha\}$$

Finally, we generalize the notion of weight to type schemes so that $\|\forall\tilde{\alpha} :: \tilde{\rho}.t\| \stackrel{\text{def}}{=} \bigvee \|t\{\tilde{s}/\tilde{\alpha}\}\|$. Note that we do not worry about instantiating $\texttt{fin}$-qualified type variables with infinite-weight types, since such type variables can be instantiated with types having arbitrarily large weight anyway. Therefore, if the weight of t depends in any way from one of the α_i, the overall weight of the type scheme will be ∞, no matter what.

We identify types modulo folding/unfolding of recursions. That is, $\texttt{rec}\ A.T = T\{\texttt{rec}\ A.T/A\}$ (we have already used this property in Definition 3.2).

4 Type System

We give the types of the constants in Table 4. The types in the l.h.s. of the table are unremarkable. The $\texttt{open}$ primitive returns a pair of peer channel endpoints

Table 4. Type of constants

$\texttt{unit} : \texttt{Unit}$	$\texttt{open} : \forall A :: \texttt{any}.\texttt{Unit} \to (A \otimes \overline{A})$
$\texttt{fix} : \forall a :: \texttt{any}.(a \to a) \to a$	$\texttt{close} : \texttt{end} \to \texttt{Unit}$
$\texttt{fork} : \texttt{Unit} \to \texttt{Unit}$	$\texttt{send} : \forall a :: \texttt{fin}.\forall A :: \texttt{any}.(a \otimes \texttt{!}a.A) \to A$
	$\texttt{receive} : \forall a :: \texttt{fin}.\forall A :: \texttt{any}.\texttt{?}a.A \to (a \otimes A)$

when applied to the `unit` value. For this reason, the resulting type is a pair of dual endpoint types. Because `open` is polymorphic, this can only be expressed using a dualized endpoint type variable. Note how `open` is an example of resource-producing function, accepting an unlimited value `unit` and returning a linear pair. The `close` primitive accepts an endpoint provided that it has type `end` and deallocates it. Being the converse of `open`, `close` is an example of resource-consuming function, accepting a linear value and not returning it. The `send` and `receive` constants implement the communication primitives: `send` accepts a message of type a, an endpoint of type $\texttt{!}a.A$ that allows sending such a message, and returns the same endpoint with the residual type A; `receive` accepts an endpoint of type $\texttt{?}a.A$, reads a message of type a from such an endpoint, and returns the pair consisting of the received message and the endpoint with the residual type A. Observe that, in both `send` and `receive`, the value type variable a is qualified by `fin`, meaning that only values with finite-weight type can be sent and received. On the contrary, no constraint is imposed on A. In the following we write $\mathsf{TypeOf}(\texttt{k})$ for the type scheme associated with `k` according to Table 4.

Judgments of the type system depend on two finite maps: the *type variable environment* $\Sigma = \{\alpha_i :: \rho_i\}_{i\in I}$ associates type variables with qualifiers, while the *name environment* $\Gamma = \{u_i : \sigma_i\}_{i\in I}$ associates names with type schemes. In both cases we use $\mathsf{dom}(\cdot)$ for denoting the set of type variables/names for which there is an association in the environment. We also write $\Sigma, \alpha :: \rho$ (respectively, $\Gamma, u : \sigma$) to extend the environment whenever $\alpha \notin \mathsf{dom}(\Sigma)$ (respectively, $u \notin \mathsf{dom}(\Gamma)$). Finally, we write $\Gamma|_{\texttt{U}}$ for the restriction of Γ to the names in `U`. Because name environments may contain linear entities (pointers) as well as unlimited ones, it is convenient to define also a more flexible (partial) operator $+$ for extending them. As in [5], we let

$$\Gamma + u : \sigma = \begin{cases} \Gamma & \text{if } u : \sigma \in \Gamma \text{ and } \mathsf{un}(\sigma) \\ \Gamma, u : \sigma & \text{if } u \notin \mathsf{dom}(\Gamma) \\ \text{undefined} & \text{otherwise} \end{cases}$$

and we extend $+$ to pairs of environments $\Gamma_1 + \Gamma_2$ by induction on Γ_2 in the natural way. We write $\mathsf{lin}(\Gamma)$ if $\mathsf{lin}(\Gamma(u))$ for some $u \in \mathsf{dom}(\Gamma)$ and $\mathsf{un}(\Gamma)$ otherwise.

Sometimes we will need to reason on the finiteness of a weight which contains type variables. In such cases, we use the information contained in a type variable environment for determining whether a weight is finite or not. More precisely, we write $\Sigma \vdash X + n < \infty$ whenever $\alpha :: \texttt{fin} \in \Sigma$ for every $\alpha \in X$.

Table 5. Typing rules for processes and expressions

(T-Thread)
$$\frac{\emptyset; \Gamma \vdash E : \texttt{Unit}}{\Gamma \vdash \langle E \rangle}$$

(T-Par)
$$\frac{\Gamma_1 \vdash P_1 \qquad \Gamma_2 \vdash P_2}{\Gamma_1 + \Gamma_2 \vdash P_1 \parallel P_2}$$

(T-Const)
$$\frac{\mathsf{un}(\Gamma) \qquad \Sigma \vdash \mathsf{TypeOf}(\texttt{k}) \succ t}{\Sigma; \Gamma \vdash \texttt{k} : t}$$

(T-Name)
$$\frac{\mathsf{un}(\Gamma) \qquad \Sigma \vdash \sigma \succ t}{\Sigma; \Gamma, u : \sigma \vdash u : t}$$

(T-Let 1)
$$\frac{\Sigma, \tilde{\alpha} :: \tilde{\rho}; \Gamma_1 \vdash E_1 : t_1 \qquad \Sigma; \Gamma_2, x : \forall \tilde{\alpha} :: \tilde{\rho}.t_1 \vdash E_2 : t_2}{\Sigma; \Gamma_1 + \Gamma_2 \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 : t_2}$$

(T-Pair)
$$\frac{\forall i \in \{1,2\} : \Sigma; \Gamma_i \vdash E_i : t_i}{\Sigma; \Gamma_1 + \Gamma_2 \vdash (E_1, E_2) : t_1 \otimes t_2}$$

(T-Let 2)
$$\frac{\Sigma; \Gamma_1 \vdash E_1 : t_1 \otimes t_2 \qquad \Sigma; \Gamma_2, x : t_1, y : t_2 \vdash E_2 : t}{\Sigma; \Gamma_1 + \Gamma_2 \vdash \texttt{let}\ x,y = E_1\ \texttt{in}\ E_2 : t}$$

(T-Arrow)
$$\frac{\Sigma; \Gamma, x : t \vdash E : s \qquad \mathsf{un}(\Gamma)}{\Sigma; \Gamma \vdash \lambda x.E : t \to s}$$

(T-Arrow Lin)
$$\frac{\Sigma; \Gamma, x : t \vdash E : s \qquad \bigvee_{u \in \mathsf{dom}(\Gamma)} \|\Gamma(u)\| \leq w}{\Sigma; \Gamma \vdash \lambda x.E : t\ w\!\multimap s}$$

(T-App)
$$\frac{\Sigma; \Gamma_1 \vdash E_1 : t \to s \qquad \Sigma; \Gamma_2 \vdash E_2 : t}{\Sigma; \Gamma_1 + \Gamma_2 \vdash E_1 E_2 : s}$$

(T-App Lin)
$$\frac{\Sigma; \Gamma_1 \vdash E_1 : t\ w\!\multimap s \qquad \Sigma; \Gamma_2 \vdash E_2 : t}{\Sigma; \Gamma_1 + \Gamma_2 \vdash E_1 E_2 : s}$$

A type scheme $\forall \tilde{\alpha} :: \tilde{\rho}.t$ denotes the family of types obtained from t by instantiating each type variable α_i with a type whose weight respects the qualifier ρ_i. This is formally expressed by an *instantiation* relation $\Sigma \vdash \sigma \succ t$ defined by the rule

$$\frac{\rho_i = \texttt{fin} \Rightarrow \Sigma \vdash \|s_i\| < \infty \ ^{(i=1..n)}}{\Sigma \vdash \forall \tilde{\alpha} :: \tilde{\rho}.t \succ t\{\tilde{s}/\tilde{\alpha}\}}$$

For example, if we consider once again the endpoint types $T_1 = \texttt{rec}\ A.!S_1.\texttt{end}$ and $S_1 = \texttt{rec}\ A.?A.\texttt{end}$ from Section 1 we have $\vdash \mathsf{TypeOf}(\texttt{send}) \succ (T_1, !T_1.\texttt{end}) \to \texttt{end}$ because T_1 has finite weight so it can instantiate the type variable a in $\mathsf{TypeOf}(\texttt{send})$. On the contrary, $\vdash \mathsf{TypeOf}(\texttt{send}) \not\succ (S_1, !S_1.\texttt{end}) \to \texttt{end}$ because $\|S_1\| = \infty$. Therefore, it is forbidden to send endpoints of type S_1.

The typing rules make use of two judgments, $\Gamma \vdash P$ stating that the process P is well typed in the name environment Γ, and $\Sigma; \Gamma \vdash E : t$ stating that E is well typed and has type t in the type variable environment Σ and name environment Γ. A judgment $\Gamma \vdash P$ is well formed if $\mathsf{dom}(\Gamma) \subseteq \mathsf{Pointers}$ and $\Gamma(\mathsf{p})$ is a closed type for every $\mathsf{p} \in \mathsf{dom}(\Gamma)$ and a judgment $\Sigma; \Gamma \vdash E$ is well formed if all type variables occurring free in Γ are in $\mathsf{dom}(\Sigma)$. Table 5 defines the typing rules for processes and expressions. Rule (T-Thread) and (T-Par) say that a process is well typed if so is each thread in it. Note that linear names are distributed linearly among threads by definition of $\Gamma_1 + \Gamma_2$. Rule (T-Const) instantiates the type of a constant, while rule (T-Name) retrieves and possibly instantiates the type of a name from the name environment. In both rules the unused part of the name environment must not contain linear resources. Rule (T-Let 1) is a linearity-aware version of the rule to have `let`-polymorphism *à la ML*. The name environment is split between E_1 and E_2 knowing that, if $\mathsf{lin}(t_1)$, then x *must* occur in E_2. Note that, by well formedness of $\Sigma, \tilde{\alpha} :: \tilde{\rho}$, none of the type variables in $\tilde{\alpha}$ can be in $\mathsf{dom}(\Sigma)$ and hence can occur free in Γ_2. Therefore, they can be safely generalized when typing E_2. Overall, this treatment of universal polymorphism is borrowed from [12]: generalization and instantiation are embedded, respectively, in rule (T-Let 1), and in rules (T-Const) and (T-Name). Rules (T-Pair) and (T-Let 2) are, respectively, the construction and the de-construction (via pattern matching) of linear pairs. Rules (T-Arrow) and (T-App) are the standard ways of introducing and eliminating (unlimited) arrow types. The rule for arrow introduction requires the side condition $\mathsf{un}(\Gamma)$, meaning that the body of the function does not make use of any pointer. Finally, rules (T-Arrow Lin) and (T-App Lin) introduce and eliminate linear arrow types. In (T-Arrow Lin), the weight w that annotates the linear arrow is chosen in such a way that it is an upper bound for the weights of the types of all names occurring in E.

Example 4.1. The following derivation, where we omit $\Sigma = a :: \texttt{fin}, A :: \texttt{any}, B :: \texttt{any}$ and we let $w = \|?a.A\| = \{a, A\}+1$, shows that the function *forward* defined at the end of Section 1 is well typed.

$$\dfrac{\dfrac{\dfrac{x : ?a.A \vdash \texttt{receive}\ x : a \otimes A \qquad \dfrac{x' : A \vdash x' : A \qquad y : !a.B, m : a \vdash \texttt{send}\ m\ y : B}{y : !a.B, m : a, x' : A \vdash (x', \texttt{send}\ m\ y) : A \otimes B}}{x : ?a.A, y : !a.B \vdash \texttt{let}\ m, x' = \texttt{receive}\ x\ \texttt{in}\ (x', \texttt{send}\ m\ y) : A \otimes B}}{x : ?a.A \vdash \lambda y.\texttt{let}\ m, x' = \texttt{receive}\ x\ \texttt{in}\ (x', \texttt{send}\ m\ y) : !a.B\ w{\multimap}\ A \otimes B}}{\vdash \lambda x.\lambda y.\texttt{let}\ m, x' = \texttt{receive}\ x\ \texttt{in}\ (x', \texttt{send}\ m\ y) : ?a.A \to !a.B\ w{\multimap}\ A \otimes B}$$

Note that w is the smallest weight allowable in this derivation. Therefore, the obtained type is also the most precise and general one for *forward*. ■

Example 4.2. In a functional language, multi-argument functions are commonly represented in *curried form*, whereby such functions accept their arguments one at a time. On the contrary, the `send` constant is *un*curried, because it accepts both its arguments at once in a pair. The *curry* combinator transforms an uncurried binary function into its curried form and is defined as *curry* $= \lambda f.\lambda x.\lambda y.f\ (x, y)$. Below is the derivation showing that *curry* is well typed, where we let $\Sigma = a :: \texttt{any}, b :: \texttt{any}, c :: \texttt{any}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Sigma; f : (a \otimes b) \to c \vdash f : (a \otimes b) \to c \qquad \dfrac{\Sigma; x : a \vdash x : a \qquad \Sigma; y : b \vdash y : b}{\Sigma; x : a, y : b \vdash (x, y) : a \otimes b}}{\Sigma; f : (a \otimes b) \to c, x : a, y : b \vdash f\ (x, y) : c}}{\Sigma; f : (a \otimes b) \to c, x : a \vdash \lambda y.f\ (x, y) : b\ \{a\}{\multimap}\ c}}{\Sigma; f : (a \otimes b) \to c \vdash \lambda x.\lambda y.f\ (x, y) : a \to b\ \{a\}{\multimap}\ c}}{\Sigma; \emptyset \vdash \lambda f.\lambda x.\lambda y.f\ (x, y) : ((a \otimes b) \to c) \to a \to b\ \{a\}{\multimap}\ c}$$

Observe that the function returned by *curry* has type $a \to b\,\{a\}\!\multimap c$, where the linear arrow type has been decorated with the weight $\{a\}$. Indeed, the function $\lambda y.f\ (x, y)$ with this type has two free variables, f having an unlimited type with null weight, and x having type a. We can now obtain the curried form of `send` as *curry* `send` which can be given the polymorphic type $\forall a :: \texttt{fin}.\forall A :: \texttt{any}.a \to\ !a.A\,\{a\}\!\multimap A$. ■

Example 4.3. The *curry* function in Example 4.2 can only be applied to functions with unlimited type. It makes sense to consider also a linear variant *lcurry* of *curry* which has the same implementation of *curry* but can be used for currying linear functions (observe that the definition of *curry* uses its first argument f exactly once). Using a derivation very similar to that shown in Example 4.2, *lcurry* could be given the type

$$\forall a :: \texttt{any}.\forall b :: \texttt{any}.\forall c :: \texttt{any}.((a \otimes b)\ w\!\multimap c) \to a\ w\!\multimap b\ (w \vee \{a\})\!\multimap c$$

except that this type depends on the weight w of the linear function being curried. This means that, in principle, we actually need a whole family lcurry_w of combinators, one for each possible weight of the linear function to be curried. However, by combining polymorphism and explicit weight annotations in linear arrow types, we can provide *lcurry* with the most general type. The idea is to introduce another type variable, say d, which does not correspond to any actual argument of the function, but which represents an arbitrary weight, and to let $w = \{d\}$. This way we can give *lcurry* the type

$$\forall a :: \texttt{any}.\forall b :: \texttt{any}.\forall c :: \texttt{any}.\forall d :: \texttt{any}.((a \otimes b)\ \{d\}\!\multimap c) \to a\ \{d\}\!\multimap b\ \{a, d\}\!\multimap c$$

where we can instantiate d with a type having exactly the weight of the linear function to be curried. For example, suppose we wish to apply *lcurry* to some function $f : (a \otimes b)\ n\!\multimap c$. Then it is enough to instantiate the d variable in the type of *lcurry* with the type $\mathtt{T}^{[n]}$ defined by

$$\mathtt{T}^{[0]} = \texttt{end} \qquad \mathtt{T}^{[m+1]} = ?\mathtt{T}^{[m]}.\texttt{end}$$

and obtain *lcurry* $f : a\ n\!\multimap b\ (\{a\} + n)\!\multimap c$ as expected. ■

Properties. In order to show that every well-typed process is well behaved (Definition 2.2) we need, as usual, a subject reduction result showing that well-typedness is preserved under reductions. Since in our language processes allocate and modify the heap, we need to define a concept of *well-typed heap* just as we have defined a concept of well-typed process. Intuitively, a heap μ is well typed with respect to an environment Γ if the endpoints allocated in μ are consistent with their type in Γ. In particular, we want that whenever a message is inserted into the queue of an endpoint, the type of the message is consistent with the type of endpoint. To this aim, we define a function $\mathsf{tail}(T, \tilde{t})$ that, given an endpoint type T and a sequence of types $\tilde{t}$ of messages, computes the residual of T after all the messages have been received:

$$\mathsf{tail}(T, \varepsilon) = T \qquad \frac{\mathsf{tail}(T, \tilde{s}) = S}{\mathsf{tail}(?t.T, t\tilde{s}) = S}$$

Note that $\mathsf{tail}(T, \tilde{s})$ is undefined if $\tilde{s}$ is not empty and T does not begin with input actions: only endpoints whose type begins with input actions can have messages in their queue. The weight of `end` and output endpoint types is zero because of this property (Definition 3.2).

The notion of well-typed heap is relative to a pair $\Gamma_0; \Gamma$ of disjoint name environments: the overall environment Γ_0, Γ determines the type of *all* the objects allocated in the heap; the sub-environment Γ distinguishes the *roots* of the heap (the pointers that are not reachable from any other pointer) from the sub-environment Γ_0 of the pointers that are stored within other structures in the heap and that are reachable from some root.

Definition 4.1 (well-typed heap). *Let* $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Gamma_0) = \emptyset$. *We say that* μ *is* well typed *in* $\Gamma_0; \Gamma$*, written* $\Gamma_0; \Gamma \Vdash \mu$*, if all of the following properties hold:*

1. *For every* $\mathsf{p} \mapsto [\mathsf{q}; \tilde{v}] \in \mu$ *we have* $\mathsf{p} \mapsto [\mathsf{q}; \tilde{w}] \in \mu$ *and either* $\tilde{v} = \varepsilon$ *or* $\tilde{w} = \varepsilon$.
2. *For every* $\mathsf{p} \mapsto [\mathsf{q}; \tilde{v}] \in \mu$ *we have* $\mathsf{tail}(T, \tilde{s}) = S$ *where* $\mathsf{p} : T \in \Gamma_0, \Gamma$ *and* $\Gamma_0|_{\mathsf{fn}(v_i)} \vdash v_i : s_i$ *and* $\|s_i\| < \infty$ *and* $\mathsf{q} \mapsto [\mathsf{p}; \varepsilon] \in \mu$ *then* $\mathsf{q} : \overline{S} \in \Gamma_0, \Gamma$.
3. $\mathsf{dom}(\mu) = \mathsf{dom}(\Gamma_0, \Gamma) = \mu\text{-}\mathsf{reach}(\mathsf{dom}(\Gamma))$.
4. *For every* $\mathtt{U}, \mathtt{V} \subseteq \mathit{dom}(\Gamma)$ *with* $\mathtt{U} \cap \mathtt{V} = \emptyset$ *we have* $\mu\text{-}\mathsf{reach}(\mathtt{U}) \cap \mu\text{-}\mathsf{reach}(\mathtt{V}) = \emptyset$.

In words, condition (1) states that in any pair of peer endpoints, one queue is always empty. This condition corresponds to *half-duplex communication*, whereby it is not possible to send messages over one endpoint before all pending messages from that endpoint have been read. Condition (2) states that the content of the queue associated with an endpoint is consistent with the type of the endpoint, that all messages in a queue have a type with finite weight, and that the residual type of an endpoint after all of the enqueued messages are received is dual of the type of its peer. Condition (3) states that all objects in the heap are reachable from the roots. Since the root pointers will be distributed linearly among the processes in the system, this means that there are no leaks. Finally, condition (4) says that every object in the heap is reachable from exactly one root, ensuring process isolation. Now we formalize the notion of well-typed system.

Definition 4.2 (well-typed system). *We say that the system* $\mu \fatsemi P$ *is* well typed *under* $\Gamma_0; \Gamma$*, written* $\Gamma_0; \Gamma \vdash \mu \fatsemi P$*, if* $\Gamma_0; \Gamma \Vdash \mu$ *and* $\Gamma \vdash P$.

We conclude this section by stating the two main results: well-typedness of systems is preserved by reductions and well-typed processes are well behaved. The proof of Theorem 4.1 relies on the finite-weight restriction on the type of messages for ensuring that no cycles are generated in the heap.

Theorem 4.1 (subject reduction). *Let* $\Gamma_0; \Gamma \vdash \mu \fatsemi P$ *and* $\mu \fatsemi P \longrightarrow \mu' \fatsemi P'$. *Then* $\Gamma_0'; \Gamma' \vdash \mu' \fatsemi P'$ *for some* Γ_0' *and* Γ'.

Theorem 4.2 (soundness). *If* $\vdash P$ *then* P *is well behaved.*

5 Related Work

This work is the convergence point of several lines of research, including the study and development of Singularity OS [9,10], the formalization of copyless messaging as a communication paradigm [1,2], the development of type systems for session-oriented functional languages [6], and polymorphic session types [4]. The fact that a linear type system is insufficient for preventing memory leaks in copyless messaging was first pointed out in [1,11]. In particular, in [1] and later in [2] we have put forward the idea of *type weight* as the characteristic quantity that allows us to discriminate between safe and unsafe messages. The main limit of the notion of type weight in [1,2] is that it is defined for endpoint types only, for which the weight is entirely determined by the structure of types. In this work we have shown that this is not always the case. Our motivation for studying the extension of the technique developed in [1,2] to a functional language is twofold: first of all, [6] already presents an elegant type system for such a language, even though [6] does not consider explicit memory management. Second, the `Sing`$^{\#}$ programming language used for the development of Singularity OS includes features such as first-class and anonymous functions, which are commonly found in functional languages. In this setting, the idea of having functions as messages turns out to be a natural one. Another major difference between the present work and [6] is that we develop a truly polymorphic type system in the style of [12], while [6] only considers monomorphic types except for communication primitives which benefit from a form of *ad hoc* polymorphism. In this sense, the present work constitutes also a smooth extension of the type system in [6] with ML-style polymorphism. Interestingly, the polymorphic type of the `open` primitive crucially relies on dualized endpoint type variables, which were introduced in [4] for totally different reasons. Note also that [6] introduces a notion of "size" for session types that may be easily confused with out notion of type weight. In [6], the size estimates the maximum number of enqueued messages in an endpoint and it is used for efficient, static allocation of endpoints with finite-size type. Our weights are unrelated to the size of queues and concern the length of chains of pointers involving queues.

6 Conclusions and Future Work

The type language we have developed is a relatively simple variant of that required for ML-style functional languages. Many features that are practically relevant can be added without posing substantial issues. For instance, it is feasible to devise a subtyping relation it in the style of [6] whereby unlimited functions can be used in place of linear ones ($t \rightarrow s \leq t \; w\!\multimap s$). Subtyping can also take into account weights, in the sense that it is safe to use a "lighter" function where a "heavier" function is expected ($t \; w\!\multimap s \leq t \; w'\!\multimap s$ if $w \leq w'$). It is also easy to equip endpoint types with the dual constructs $T \oplus S$ and $T + S$ for denoting internal and external choices driven by boolean values.

The finite-weight restriction on the type of messages prevents the formation of cycles in the heap. In the context of Singularity OS, this restriction seems to be reasonable since objects allocated in the exchange heap are managed by means of reference counting which cannot handle cyclic structures.

The type system we have presented (Table 5) is not syntax-directed and therefore leaves room for a fair amount of "guessing", in particular with respect to the introduction of type variables in types and weights. An open question is whether it is feasible to devise a fully automated type and weight inference algorithm that is capable of inferring the most general type of arbitrary expressions.

Acknowledgments. This work has been partially supported by MIUR PRIN 2010-2011 CINA. The authors are grateful to the anonymous referees for their comments.

References

1. Bono, V., Messa, C., Padovani, L.: Typing Copyless Message Passing. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 57–76. Springer, Heidelberg (2011)
2. Bono, V., Padovani, L.: Typing Copyless Message Passing. Logical Methods in Computer Science 8, 1–50 (2012)
3. Bono, V., Padovani, L., Tosatto, A.: Polymorphic Types for Leak Detection in a Session-Oriented Functional Language (2013), `http://www.di.unito.it/~padovani/Papers/BonoPadovaniTosatto13.pdf`
4. Gay, S.: Bounded Polymorphism in Session Types. Mathematical Structures in Computer Science 18(5), 895–930 (2008)
5. Gay, S., Hole, M.: Subtyping for Session Types in the π-calculus. Acta Informatica 42(2-3), 191–225 (2005)
6. Gay, S., Vasconcelos, V.T.: Linear Type Theory for Asynchronous Session Types. Journal of Functional Programming 20(01), 19–50 (2010)
7. Honda, K.: Types for Dyadic Interaction. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 509–523. Springer, Heidelberg (1993)
8. 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. 122–138. Springer, Heidelberg (1998)
9. Hunt, G., Larus, J., Abadi, M., Aiken, M., Barham, P., Fähndrich, M., Hawblitzel, C., Hodson, O., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D., Wobber, T., Zill, B.: An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research (2005)
10. Hunt, G.C., Larus, J.R.: Singularity: Rethinking the Software Stack. SIGOPS Operating Systems Review 41, 37–49 (2007)
11. Villard, J.: Heaps and Hops. PhD thesis, Laboratoire Spécification et Vérification, ENS Cachan, France (2011)
12. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation 115(1), 38–94 (1994)

Passive Testing with Asynchronous Communications[★]

Robert M. Hierons[1], Mercedes G. Merayo[2], and Manuel Núñez[2]

[1] Department of Information Systems and Computing, Brunel University
Uxbridge, Middlesex, UB8 3PH United Kingdom
rob.hierons@brunel.ac.uk

[2] Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Madrid, Spain
mgmerayo@fdi.ucm.es, mn@sip.ucm.es

Abstract. Testing is usually understood to involve the tester interacting with the studied system by supplying input and observing output. However, sometimes this *active* interaction is not possible and testing becomes more *passive*. In this setting, passive testing can be considered to be the process of checking that the observations made regarding the system satisfy certain required properties. In this paper we study a formal passive testing framework for systems where there is an asynchronous communications channel between the tester and the system. We consider a syntactic definition of a class of properties and provide a semantic representation, as automata, that take into account the different observations that we can expect due to the assumption of asynchrony. Our solution checks properties against traces in polynomial time, with a low need for storage. Therefore, our proposal is very suitable for real-time passive testing.

1 Introduction

Testing is widely used to increase the reliability of complex systems. Traditionally, testing of software had a weak formal basis, in contrast with testing of hardware systems where different formalisms and formal notations were used from the beginning [23,10]. However, it has been recognised that *formalising* the different aspects of testing of software is very beneficial [7]. The combination of formal methods and testing is currently well understood, tools to automate testing activities are widely available, there are several surveys on the field [14,13], and industry is becoming aware of the importance of using formal approaches [8].

Usually, testing consists of applying stimuli (inputs) to the system and deciding whether the observed reactions (outputs) are those expected. However, we might be required to assess a particular system but without having access to it. The reasons for these limitations might be due to security issues or because the

[★] Research partially supported by the Spanish projects TESIS and ESTuDIo (TIN2009-14312-C02 and TIN2012-36812-C02) and the UK EPSRC project Testing of Probabilistic and Stochastic Systems (EP/G032572/1).

system is running 24/7 and our interaction might produce undesirable changes in the associated data. In these situations, testing can still play a role, but becomes more *passive* since interaction will be replaced by observation. Formal passive testing is already a well established line of research and extensions of the original frameworks [21,4,2] have dealt with issues such as security and time [24,22,1]. Essentially, in passive testing we have a property and we check that the trace being observed satisfies that property. Ideally, we want the process of checking whether the property is satisfied to be quick and to take very little storage since this can allow passive testing to occur in real-time. The application of passive testing in real-time has an important benefit: a detected error can be notified to the operators of the system almost immediately and they can then take appropriate measures. If the trace needs to be saved and processed off-line, then the time between the detection of the error and the corresponding notification will significantly increase.

One possible approach to work with properties and traces is to represent the property P as an automaton $M(P)$ such that a trace satisfies P if and only if it is not a member of the language defined by $M(P)$: it does not reach a final (error) state of $M(P)$. If $M(P)$ is deterministic then the process of checking whether a trace satisfies P takes linear time and is an incremental process: every time we observe a new input or output we simply update the state of $M(P)$. Even if $M(P)$ is non-deterministic, the process of checking whether a trace satisfies P takes time that is linear in the size of $M(P)$ and the length of the trace and the process is still incremental. This makes such an automaton based approach desirable if we can find efficient ways of mapping properties to automata.

Previous work on passive testing has assumed that the monitor that observes and checks the behaviour of the System Under Test (SUT) observes the actual trace produced. However, the monitor might not directly observe the interface of the SUT and instead there may be an asynchronous channel/network between the monitor and the SUT. Where this is the case, the trace observed by the monitor might not be the one produced by the SUT: input is observed before it is received by the SUT and output is observed after it is sent by the SUT. Suppose, for example, that the monitor observes the trace $?i?i!o$ in which $?i$ is an input and $!o$ is an output. In this case it is possible that the SUT actually produced either $?i!o?i$ or $!o?i?i$ and that the observation of $?i?i!o$ was due to the delaying of output. Thus, if we have properties that the SUT should satisfy and we directly apply them in such a context then we may obtain false positives or false negatives. We therefore require new approaches to passive testing in such circumstances and in this paper we focus on the case where the asynchronous channels are first in first out (FIFO).

There are several ways of applying passive testing when observations are through FIFO channels. One approach creates a model of our property and adds queues to this. However, the addition of queues can lead to the model requiring more storage space. In particular, if the queues are not bounded then it leads to there being an infinite number of states while if there is a bound on the queue length then the number of states increases exponentially with this bound.

An alternative is to transform the trace ρ observed to form an automaton M_ρ that represents all traces that might lead to ρ being observed where there is asynchronous communications. However, under this approach we have to analyse the entire, potentially very long, trace that has been observed and the monitor has to store this. This approach thus mitigates against real-time uses since it can significantly increase the storage and processing requirements. In this paper we explore a third alternative, in which for a property P we produce an automaton $\mathcal{A}(P)$ such that we check whether a trace ρ observed is a member of the language defined by $\mathcal{A}(P)$.

We are not aware of previous work on passive testing where there is an asynchronous communications channel between the system and the monitor. In contrast, there has been some work on active testing and several approaches have appeared in the literature for models where there is a distinction between inputs and outputs [11,20,25,12]. Passive testing is a monitoring technique and as such it is related to runtime verification since they share the same goal, checking the correctness of a system without interacting with it, but use different formalisms and methodologies. In runtime verification it is not usual to distinguish between inputs and outputs, since their observation makes them events of the same nature, and therefore it is difficult to compare the work from that area with ours. While some work has investigated asynchronous runtime monitoring, the problems considered in this context are different: this line of work does not distinguish between input and output and does not explore potential reorderings of traces. Instead, it looks at the situation in which the monitor and system do not synchronise on actions: actions engaged in by the system might instead be recorded and analysed later, with a compensation phase being used to undo any later actions if an error is found [5].

The rest of the paper is structured as follows. In Section 2 we introduce notation to define systems and traces that will be used throughout the paper. Section 3 introduces the notion of an ideal that will be used in creating automata from observations. Section 4 is the bulk of the paper and presents how properties can be translated into automata and provides our theory to check traces against properties. Finally, in Section 5 we present our conclusions and provide some lines for future work.

2 Preliminaries: Systems and Observations

In this section we introduce the basic notion used in this paper to define systems as well as concepts associated with the traces that a system can perform and with the traces that a monitor can actually observe in an asynchronous setting.

Definition 1. *An* input-output transition system (IOTS) $M = (Q, I, O, T, q_{in})$ *is a tuple in which Q is a countable set of states, $q_{in} \in Q$ is the initial state, I is a countable set of inputs, O is a countable set of outputs, and $T \subseteq Q \times (I \cup O) \times Q$ is the transition relation. A transition $(q, a, q') \in T$ means that from state q it is possible to move to state q' with action $a \in I \cup O$. We use the following notation concerning the performance of (sequences of) actions.*

- *$\mathcal{A}ct = I \cup O$ is the set of actions.*
- *If $(q, a, q') \in T$, for $a \in \mathcal{A}ct$, then we write $q \xrightarrow{a} q'$ and $q \xrightarrow{a}$.*
- *We write $q \overset{\sigma}{\Longrightarrow} q'$ for $\sigma = a_1 \dots a_m \in \mathcal{A}ct^*$, with $m \geq 0$, if there exist $q_0, \dots, q_m$, $q = q_0$, $q' = q_m$ such that for all $0 \leq i < m$ we have that $q_i \xrightarrow{a_{i+1}} q_{i+1}$. Note that $q \overset{\epsilon}{\Longrightarrow} q$, where ϵ is the empty sequence.*
- *If there exists q' such that $q_{in} \overset{\sigma}{\Longrightarrow} q'$ then we say that σ is a* trace *of M and we write $M \overset{\sigma}{\Longrightarrow}$. We let $L(M)$ denote the set of traces of M.*

We have an asynchronous setting and, therefore, we do not have to consider only the traces that can be performed by a system but also how these traces can be observed. Intuitively, if a system performs a certain trace then we can observe a variation of this trace where the outputs appear later than they were actually performed. Next we formally define this idea and given a system M and a trace σ we let $\mathcal{L}(\sigma)$ denote the set of traces that might be observed by a monitor if M produces trace σ and communications between the monitor and the SUT are asynchronous and FIFO.

Definition 2. *Let I and O be sets of inputs and outputs, respectively, and $\sigma, \sigma' \in \mathcal{A}ct^*$ be sequences of actions. We say that σ' is an observation of σ, denoted by $\sigma \rightsquigarrow \sigma'$, if there exist sequences $\sigma_1, \sigma_2 \in \mathcal{A}ct^*$, $!o \in O$ and $?i \in I$ such that $\sigma = \sigma_1 !o ?i \sigma_2$ and $\sigma' = \sigma_1 ?i !o \sigma_2$. We let $\mathcal{L}(\sigma)$ denote the set of traces that can be formed from σ through sequences of transformations of the form $\rightsquigarrow$, that is, $\mathcal{L}(\sigma) = \{\sigma' | \sigma \rightsquigarrow^* \sigma'\}$, where $\rightsquigarrow^*$ represents the repeated application of $\rightsquigarrow$. We overload this to say that given an IOTS M, $\mathcal{L}(M) = \cup_{\sigma \in L(M)} \mathcal{L}(\sigma)$ is the set of traces that might be observed when interacting with M through asynchronous FIFO channels.*

Example 1. Assume that the SUT has produced the trace $\sigma = ?i_1 !o_1 !o_2 ?i_2 !o_1$. Due to the asynchronous nature of the system, the monitor might observe any of the traces in the set $\mathcal{L}(\sigma) = \{?i_1 !o_1 !o_2 ?i_2 !o_1, ?i_1 !o_1 ?i_2 !o_2 !o_1, ?i_1 ?i_2 !o_1 !o_2 !o_1\}$.

3 Sets of Events from Observations and Ideals

In line with previous work in formal passive testing [2], we will consider properties of the form (σ, O_σ) for $\sigma \in \mathcal{A}ct^*$ and $O_\sigma \subseteq O$. Such a property says that if the SUT produces the sequence σ then the next output must from the set O_σ. It is straightforward to devise an automaton $M(P)$ that accepts only the traces that do not satisfy such a property P: we define $M(P)$ such that it accepts the regular language $\mathcal{A}ct^* \{\sigma\} (\mathcal{A}ct \setminus O_\sigma) \mathcal{A}ct^*$. It is easy to check that $M(P)$ accepts a trace ρ if and only if ρ does not satisfy this property: ρ contains a subsequence that has σ followed by an action that is not in O_σ. It is well known that an automaton that represents a regular expression can be produced in quadratic time [3]. This process can be further improved to achieve sub-quadratic complexity and can be efficiently parallelised to work in $O(\log(|\sigma|))$ time [9]. Such an automaton $M(P)$ has $O(|\sigma|)$ states and $O(|\sigma| \cdot \log(|\sigma|)^2)$ transitions [18]. In the next section we adapt the above approach for the case where communications are asynchronous.

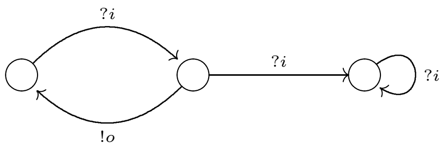


Fig. 1. An IOTS M such that $\mathcal{L}(M)$ is not regular

Before outlining our solution, we briefly comment on some alternatives. Previous work has described a delay operator that takes a trace σ of an IOTS and returns the set of traces that might be observed if the SUT produces σ and interacts asynchronously through FIFO channels with its environment [19]. However, the delay operator cannot be applied directly since it applies to a single trace rather than an automaton $M(P)$. While it has been shown how a test purpose can be adapted to incorporate the delay into the verdict [25], this approach assumes that the tester waits for output before sending the next input and so does not apply input in a state where output can be produced. Since the input is not supplied by the tester in passive testing, we cannot make such an assumption. We might instead aim to define a general method that takes an IOTS M (with finite sets of states and transitions) and produces IOTS M' (with finite sets of states and transitions) with $L(M') = \mathcal{L}(M)$. If we can achieve this then M' can be used. However, the following shows that there is no such general method.

Proposition 1. *Given an IOTS M with finite sets of states and transitions, there may be no IOTS M' with finite sets of states and transitions such that $L(M') = \mathcal{L}(M)$.*

Proof. An IOTS with finite sets of states and transitions defines a regular language so it is sufficient to find some such M where $\mathcal{L}(M)$ is not a regular language. Let M be the IOTS with three states shown in Figure 1 (the initial state is represented by the leftmost vertex). We will use proof by contradiction, assuming that $\mathcal{L}(M)$ is a regular language. Thus, since $\mathcal{L}(M)$ is regular and I^*O^* is regular we have that $\mathcal{L}(M) \cap (I^*O^*)$ is regular. However, $\mathcal{L}(M) \cap (I^*O^*)$ contains all sequences of the form of n inputs followed by n or fewer outputs and this is not a regular language. This provides a contradiction as required.

While this result shows that there is no general method that takes a property P defined by an IOTS $M(P)$ with finite sets of states and transitions and returns a suitable property for use when communications are asynchronous, we will see in the next section that we can take advantage of the structure of the properties we consider. First we will show how, for trace σ, we can produce an automaton $\mathcal{A}^T(\sigma)$ that gives the set of traces that might be observed if the SUT produces σ. Since we are applying passive testing, a trace σ of interest might not be the start of the overall trace observed and so we will then adapt $\mathcal{A}^T(\sigma)$ to produce the automaton $\mathcal{A}(\sigma, O_\sigma)$ that will be used.

Given a sequence σ, we will define a partial order $\ll$ on the inputs and outputs in σ to represent which actions must be performed before other ones if the SUT produces σ. In order to distinguish between repeated actions in the trace, events are constructed from actions by labelling each action in σ with the occurrence of the symbol in the trace.

Definition 3. *Let $\sigma = a_1 \dots a_n \in \mathcal{A}ct^*$ be a sequence of actions. We let $E(\sigma)$ denote the set of events of σ, where $e = (a_i, k)$ belongs to $E(\sigma)$ if and only if there are exactly $k-1$ occurrences of a_i in $a_1 \dots a_{i-1}$. This says that the ith element of σ is the kth instance of a_i in σ.*

Example 2. Consider the trace $\sigma = ?i_1!o_1!o_2?i_2!o_1$. The corresponding set of events is $E(\sigma) = \{(?i_1, 1), (!o_1, 1), (!o_2, 1), (?i_2, 1), (!o_1, 2)\}$.

Definition 4. *Let $\sigma \in \mathcal{A}ct^*$ be a sequence of actions. Given two events $e_i = (a_i, k_i)$ and $e_j = (a_j, k_j)$ belonging to $E(\sigma)$, we write $e_i \ll e_j$ if either $i = j$ or $i < j$ and one of the following conditions hold: a_i and a_j are inputs, or a_i and a_j are outputs, or a_i is an input and a_j is an output.*

The first two cases in the definition of $\ll$ result from channels being FIFO. The last case results from the observation of outputs being delayed, while an input is observed before it is received by the SUT. Essentially, $(a_i, k_i) \ll (a_j, k_j)$ does not hold for $i < j$ if a_i is an output and a_j is an input since in this case it is possible that the observation of output a_i is delayed until after input a_j has been sent. Given a trace $\sigma \in \mathcal{A}ct^*$ it is straightforward to prove that $(E(\sigma), \ll)$ is a partially ordered set. Next we introduce the notions of an *ideal* and anti-chains of a set of events.

Definition 5. *Let $\sigma \in \mathcal{A}ct^*$ be a sequence of actions and $E(\sigma)$ be the set of its events, possibly annotated to avoid repetitions. A set $\mathcal{I} \subseteq E(\sigma)$ is said to be an* ideal *of $(E(\sigma), \ll)$ if for all $e_i, e_j \in E(\sigma)$, if $e_i \ll e_j$ and $e_j \in \mathcal{I}$ then $e_i \in \mathcal{I}$. An ideal $\mathcal{I}$ is a* principal ideal *if there is some e_j such that $\mathcal{I}$ contains only e_j and all elements below it under $\ll$, that is, $\mathcal{I} = \{e_i \in E(\sigma) | e_i \ll e_j\}$. Finally, a set $E' \subseteq E(\sigma)$ is an* anti-chain *if no two different elements of E' are related under $\ll$.*

The essential idea is that if the SUT produces σ and e_i is a maximal element of ideal $\mathcal{I}$, then $\mathcal{I}$ includes all events that *must* be observed before e_i is observed by the monitor.

Example 3. Consider again the trace $\sigma = ?i_1!o_1!o_2?i_2!o_1$. We have that the following sets of events $\mathcal{I}_1 = \{(?i_1, 1), (!o_1, 1), (!o_2, 1)\}$, $\mathcal{I}_2 = \{(?i_1, 1), (?i_2, 1)\}$ and $\mathcal{I}_3 = \{(?i_1, 1), (!o_1, 1), (?i_2, 1)\}$ are ideals of $(E(\sigma), \ll)$. However, only $\mathcal{I}_1$ and $\mathcal{I}_2$ are principal ideals. The ideal $\mathcal{I}_3$ contains the events $(!o_1, 1)$ and $(?i_2, 1)$ that are not related under $\ll$, therefore, $\mathcal{I}_3$ is not a principal ideal. Finally, the sets $E_1 = \{(!o_1, 1), (?i_2, 1)\}$ and $E_2 = \{(!o_2, 1), (?i_2, 1)\}$ are anti-chains of $(E(\sigma), \ll)$.

Next we present an alternative characterisation of the notion of ideal that shows that an ideal is defined by its maximal (under $\ll$) elements.

Lemma 1. *Let $\sigma \in \mathcal{A}ct^*$ be a sequence of actions. We have that $\mathcal{I} \subseteq E(\sigma)$ is an ideal if and only if one of the following conditions holds:*

- *$\mathcal{I}$ contains an input a_i and all earlier inputs;*
- *$\mathcal{I}$ contains an output a_j and all earlier inputs and outputs; or*
- *$\mathcal{I}$ contains an input a_i, an output a_j, all inputs before a_i, and all inputs and outputs before a_j.*

The following classical result [6] relates ideals and anti-chains.

Proposition 2. *The set of ideals is isomorphic to the set of anti-chains, by associating with every anti-chain E' the ideal which is the union of the principal ideals generated by the elements of E'. Vice versa, the anti-chain corresponding to a given ideal $\mathcal{I}$ is the set of maximal elements of $\mathcal{I}$.*

The following result provides a measure, in the worst case, on the number of ideals contained in a set of events. This result will be relevant since it will be used to calculate the complexity of the algorithm that computes the automaton associated with a certain property P.

Proposition 3. *Let $\sigma \in \mathcal{A}ct^*$ be a sequence of actions with length m. There are $O(m^2)$ ideals in $E(\sigma)$.*

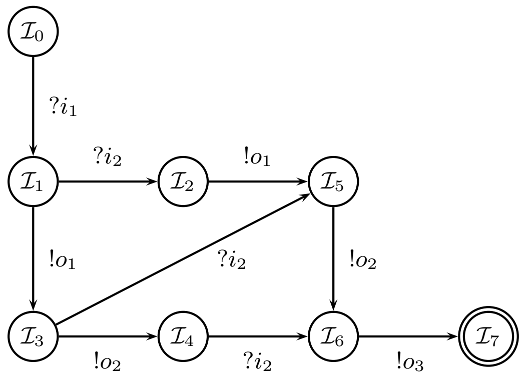
Proof. By Proposition 2 we know that the number of ideals is the same as the number of anti-chains. However, we also know that any two inputs in $E(\sigma)$ are related under $\ll$. Similarly, any two outputs in $E(\sigma)$ are related under $\ll$. Thus, an anti-chain can have at most two elements (one input and one output) and so there are $O(m^2)$ anti-chains. The result therefore holds.

An ideal $\mathcal{I}$ is a set of elements from $E(\sigma)$ such that all 'earlier' elements, under $\ll$, are contained in $\mathcal{I}$. Ideal $\mathcal{I}$ of $(E(\sigma), \ll)$ is therefore one possible set of events that might be observed, as the prefix of a trace from $\mathcal{L}(\sigma)$, if the SUT produces σ. Thus, an ideal $\mathcal{I}$ defines a set of events in one or more prefixes of a trace from $\mathcal{L}(\sigma)$. Similarly, the events in a prefix of a trace from $\mathcal{L}(\sigma)$ form an ideal of $E(\sigma, \ll)$. As a result, we can reason about prefixes of traces in $\mathcal{L}(\sigma)$ by considering the ideals of $(E(\sigma), \ll)$.

4 Creating Automata for Properties

In this section we show how the ideals associated with a certain trace can be used to construct appropriate automata. More specifically, given a sequence of actions σ, we will use the ideals of $(E(\sigma), \ll)$ to represent states of a finite automaton $\mathcal{A}^T(\sigma)$ that accepts the set of sequences in $\mathcal{L}(\sigma)$. We will study properties of these automata and the time complexity of using them in passive testing.

Definition 6. *Given non-empty $\sigma \in \mathcal{A}ct^*$ we let $\mathcal{A}^T(\sigma)$ denote the finite automaton with state set S that is equal to the set of ideals of $(E(\sigma), \ll)$, alphabet $\mathcal{A}ct$, initial state $\{\}$ and the following set of transitions: given ideal $\mathcal{I}$ and $a \in \mathcal{A}ct$, there is a transition $t = (\mathcal{I}, a, \mathcal{I}')$ for ideal $\mathcal{I}'$ if and only if $\mathcal{I}' = \mathcal{I} \cup \{a\}$. In addition, $\mathcal{A}^T(\sigma)$ has one final state, which is the ideal $E(\sigma)$.*



$$
\begin{aligned}
\mathcal{I}_0 &= \{\} \\
\mathcal{I}_1 &= \{?i_1\} \\
\mathcal{I}_2 &= \{?i_1, ?i_2\} \\
\mathcal{I}_3 &= \{?i_1, !o_1\} \\
\mathcal{I}_4 &= \{?i_1, !o_1, !o_2\} \\
\mathcal{I}_5 &= \{?i_1, !o_1, ?i_2\} \\
\mathcal{I}_6 &= \{?i_1, !o_1, !o_2, ?i_2\} \\
\mathcal{I}_7 &= \{?i_1, !o_1, !o_2, ?i_2, !o_3\}
\end{aligned}
$$

Fig. 2. Automaton $\mathcal{A}^T(\sigma)$ for the trace $\sigma = ?i_1!o_1!o_2?i_2!o_3$

A finite automaton A is essentially an IOTS with finite sets of states, inputs and outputs and a set of final states. Then A defines the regular language $L(A)$ of labels of walks from the initial state of A to final states of A. Note that an IOTS with finite sets of states and actions can be seen as a finite automaton where all the states are final.

Example 4. Let $\sigma = ?i_1!o_1!o_2?i_2!o_3$ be a trace. Figure 2 depicts the automaton $\mathcal{A}^T(\sigma)$ that accepts the set of sequences in $\mathcal{L}(\sigma)$.

We have that $\mathcal{A}^T(\sigma)$ defines the set of behaviours, $\mathcal{L}(\sigma)$, that can be observed if the SUT produces σ.

Proposition 4. *Given $\sigma \in \mathcal{A}ct^*$ we have that $L(\mathcal{A}^T(\sigma)) = \mathcal{L}(\sigma)$.*

Proof. We will prove a slightly stronger result, which is that ρ labels a walk from the initial state of $\mathcal{A}^T(\sigma)$ if and only if ρ is a prefix of a sequence in $\mathcal{L}(\sigma)$.

We first prove the left to right implication by induction on the length of ρ. The result clearly holds for the base case in which ρ is the empty sequence. Now assume that it holds for all sequences of length less than k, $k \geq 1$, and ρ has length k. Thus, $\rho = \rho_1 a$ for some $a \in I \cup O$. By the inductive hypothesis we have that ρ_1 is a prefix of a sequence in $\mathcal{L}(\sigma)$. In addition, by the definition of $\mathcal{A}^T(\sigma)$, we have that the set of events in ρ_1 forms an ideal $\mathcal{I}_1$ and $\mathcal{I}_1 \cup \{a\}$ is an ideal. Thus, since $\mathcal{I}_1 \cup \{a\}$ is an ideal, there does not exist $b \in E(\sigma) \setminus (\mathcal{I}_1 \cup \{a\})$ such that $b \ll a$. By the definition of $\mathcal{L}(\sigma)$ we have that ρ_1 can be followed by a and so $\rho_1 a$ is a prefix of a sequence in $\mathcal{L}(\sigma)$ as required.

We now prove the right to left implication, again, by induction on the length of ρ. The result clearly holds for the base case in which ρ is the empty sequence. Now assume that it holds for all sequences of length less than k, $k \geq 1$, and ρ has length k. Thus, $\rho = \rho_1 a$ for some $a \in I \cup O$. By the inductive hypothesis we have that ρ_1 is the label of a walk of $\mathcal{A}^T(\sigma)$ and assume that this walk reaches a state representing ideal $\mathcal{I}_1$. By the definition of $\mathcal{L}(\sigma)$ there cannot exist an action in σ that is not in ρ_1 and that must be observed before a and so precedes a under $\ll$.

Algorithm 1. Producing $\mathcal{A}(\sigma, O_\sigma)$

```
1: Input $(\sigma, O_\sigma)$.
2: Let $\mathcal{A}(\sigma, O_\sigma) = \mathcal{A}^T(\sigma)$, let $s_0$ denote the initial state of $\mathcal{A}(\sigma, O_\sigma)$, and let $s_f$ denote
   the final state of $\mathcal{A}(\sigma, O_\sigma)$.
3: For all $a \in \mathcal{A}ct$ add the transition $(s_0, a, s_0)$. These transitions ensure that we are
   considering all possible starting points in a trace $\rho'$ observed.
4: For every state $s$ of $\mathcal{A}(\sigma, O_\sigma)$ that represents an ideal that does not contain output
   and for all $!o \in O$, add the transition $(s, !o, s)$. These transitions correspond to the
   possibility of earlier output being observed after input from $\sigma$.
5: For every state $s$ of $\mathcal{A}(\sigma, O_\sigma)$ that represents an ideal that contains all of the input
   from $\sigma$ and for all $?i \in I$, add the transition $(s, ?i, s)$. These transitions correspond
   to the possibility of later input being observed before some of the output from $\sigma$.
6: Add a new state $s_e$ to $\mathcal{A}(\sigma, O_\sigma)$ and for all $!o \in O \backslash O_\sigma$ add the transition $(s_f, !o, s_e)$.
   If we have observed the input and output from $\sigma$ and the next output is not from
   $O_\sigma$ then go to the final (error) state.
7: Make $s_e$ the only final state of $\mathcal{A}(\sigma, O_\sigma)$.
8: Complete $A$: if there is no transition from a state $s \neq s_e$ with label $a \in \mathcal{A}ct$ then
   add the transition $(s, a, s_0)$.
9: Output $\mathcal{A}(\sigma, O_\sigma)$.
```

Thus, $\mathcal{I}_2 = \mathcal{I}_1 \cup \{a\}$ is an ideal and so $\mathcal{A}^T(\sigma)$ contains a transition from the state representing $\mathcal{I}_1$ to the state representing $\mathcal{I}_2$ with label a, concluding that $\rho = \rho_1 a$ is the label of a walk of $\mathcal{A}^T(\sigma)$ as required. The result therefore follows.

We now have to adapt $\mathcal{A}^T(\sigma)$ to take into account two points: σ might be preceded by other actions and the observation of earlier outputs might be delayed; and σ might be followed by later actions and the outputs from σ might not be observed until after later inputs. Algorithm 1 achieves this.

Example 5. Let $\sigma = ?i_1!o_1!o_2?i_2!o_3$ and consider the automaton $\mathcal{A}^T(\sigma)$ depicted in Figure 2. Given a set of outputs O_σ, Figure 3 shows the automaton $\mathcal{A}(\sigma, O_\sigma)$ constructed by using Algorithm 1.

Before proving that Algorithm 1 returns the correct result, we define what it means for an automaton A to be sound: if the SUT produces a trace that does not satisfy property P then the trace observed by the monitor is in $L(A)$.

Definition 7. *Let P be a property and A be a finite automaton. We say that A is* sound for P *if and only if whenever the SUT produces a trace σ_1 that does not satisfy property P and the trace $\sigma'_1 \in \mathcal{L}(\sigma_1)$ is observed we have that $\sigma'_1 \in L(A)$.*

This essentially corresponds to the automaton not being able to produce false positives: if the SUT fails property P, then the automaton will produce an 'alarm'. The automaton produced by Algorithm 1 is sound.

Theorem 1. *Given property $P = (\sigma, O_\sigma)$, the automaton $\mathcal{A}(P)$ returned by Algorithm 1 when given P is sound for P.*

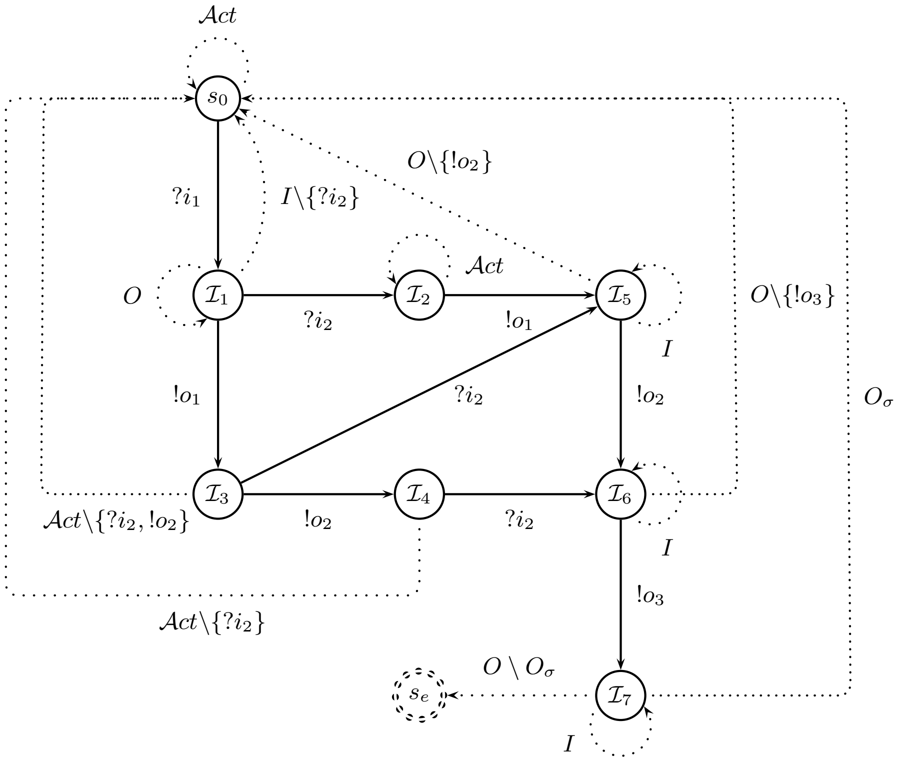


Fig. 3. Automaton $\mathcal{A}(\sigma, O_\sigma)$ for the trace $\sigma = ?i_1!o_1!o_2?i_2!o_3$ and a set of outputs O_σ

Proof. Recall that we label actions using their occurrence, if necessary, so that they are unique and this labelling is preserved by the delay of output. We assume that the SUT has produced a trace σ_1 that does not satisfy P, that this led to the observation of the trace $\sigma_1' \in \mathcal{L}(\sigma_1)$ and we are required to prove that $\sigma_1' \in L(\mathcal{A}(\sigma, O_\sigma))$. Since σ_1 does not satisfy P we have that $\sigma_1 = \sigma_2\sigma a\sigma_3$ for some $a \in O \setminus O_\sigma$. Since $\sigma_1' \in \mathcal{L}(\sigma_1)$ we have that $\sigma_1' = \sigma_2'\sigma' a\sigma_2'$ for some $\sigma', \sigma_2', \sigma_3'$ such that σ' satisfies the following.

– σ' starts with the first action of σ_1' that is from σ;
– σ' may contain outputs not in σ (delayed from σ_2);
– σ' may contain inputs not in σ (due to outputs from σ or a being delayed past inputs from σ_3); and
– σ' can have outputs from σ being delayed past inputs from σ.

By the definition of $\mathcal{A}(\sigma, O_\sigma)$ we have that the state of $\mathcal{A}(\sigma, O_\sigma)$ after σ_1' can be the initial state of $\mathcal{A}(\sigma, O_\sigma)$. Further, by Proposition 4 we know that σ' with the extra initial outputs and final inputs removed can take $\mathcal{A}(\sigma, O_\sigma)$ to the final state s_f. Consider the corresponding path ρ of $\mathcal{A}(\sigma, O_\sigma)$.

The additional outputs in σ' that are not in σ (and so come from σ_2) are all before the first output in σ' that was from σ and so, by construction, we can define a path ρ' that includes these by adding self-loops to ρ.

Similarly, the additional inputs in σ' that are not in σ (and so come from σ_3) are all after the last input in σ' that was from σ and so we can define a path ρ'' that includes these by adding self-loops to ρ'. Path ρ'' thus takes $\mathcal{A}(\sigma, O_\sigma)$ to state s_f and has label σ'. The result now follows from observing that a takes $\mathcal{A}(\sigma, O_\sigma)$ from state s_f to the final state and this final state cannot be left.

An automaton A being sound for P denotes an absence of false positives. However, we might also want the absence of false negatives: if the SUT produces a trace and the resultant observation is in $L(A)$ then the trace produced by the SUT must not have satisfied P. This is captured by the notion of *exact*.

Definition 8. *Let P be a property and A be a finite automaton. We say that A is* exact *for P if and only if whenever the SUT produces some trace σ_1 and the observed trace $\sigma_1' \in \mathcal{L}(\sigma_1)$ is such that $\sigma_1' \in L(A)$ we must have that σ_1 does not satisfy P.*

The automaton $\mathcal{A}(\sigma, O_\sigma)$, produced by our algorithm, need not be exact for (σ, O_σ) as the following example shows.

Example 6. Consider the property $P = (?i, \{!o\})$ and the observed trace $\sigma_1' = ?i!o'!o$ which is in the language defined by the automaton $\mathcal{A}(?i, \{!o\})$. We can consider two examples for the trace σ_1 produced by the SUT.

– $\sigma_1 = !o'?i!o$ and so σ_1 satisfies P.
– $\sigma_1 = ?i!o'!o$ and so σ_1 does not satisfy P.

The above shows that an observation made might be consistent with both traces that satisfy P and traces that do not. Therefore, our automata might not be exact. Note that this is not a drawback of our theory, but a consequence of working within a framework where the available information does not allow us to reconstruct the trace that was originally performed by the system. Similar situations appear in other frameworks such as when testing systems with distributed interfaces [15,16]. We are currently working on approaches that allow us to make stronger statements regarding the failure of a property. We discuss our preliminary ideas in the part of the next section devoted to future work.

Even though we cannot expect *exactitude*, we should be able to ensure that if the observed trace is one that might have resulted from a trace of the SUT that does not satisfy the property P then the observed trace is in $L(A)$. This is captured by the following notion.

Definition 9. *Let P be a property and A be a finite automaton. We say that A is* precise *for P if and only if whenever a trace σ_1' is in $L(A)$ there is some trace σ_1 that does not satisfy P such that $\sigma_1' \in \mathcal{L}(\sigma_1)$.*

The following result shows that Algorithm 1 returns an automaton that is precise for the considered property.

Theorem 2. *Given property $P = (\sigma, O_\sigma)$, the automaton $\mathcal{A}(P)$ returned by Algorithm 1 when given P is precise for P.*

Proof. Suppose that trace σ_1' is in $L(\mathcal{A}(\sigma, O_\sigma))$. By definition it is sufficient to prove that there exists some trace σ_1 that does not satisfy P such that $\sigma_1' \in \mathcal{L}(\sigma_1)$. Note that σ_1 does not satisfy P if and only if it has a prefix that ends in σa for some $a \in O \setminus O_\sigma$.

By the construction of $\mathcal{A}(\sigma, O_\sigma)$, since $\sigma_1' \in L(\mathcal{A}(\sigma, O_\sigma))$, we have that σ_1' has prefix $\sigma_2'\sigma_3'a$ such that σ_3' takes $\mathcal{A}(\sigma, O_\sigma)$ from state s_0 to s_f and $a \in O \setminus O_\sigma$. In addition, we must have that σ_3' differs from σ in only three ways:

– the addition of outputs before the outputs of σ, through self-loops in states that correspond to ideals that contain no output;
– the addition of inputs after the inputs of σ, through self-loops in states that correspond to ideals that contain all of the inputs from σ; and
– the delay in output from σ.

Thus, the input projection of σ_3' is the input projection of σ followed by some sequence σ_I' of inputs and the output projection of σ_3' is some sequence σ_O' of outputs followed by the output projection of σ. As a result, $\sigma_3'a \in \mathcal{L}(\sigma_O'\sigma\sigma_I'a)$. Thus, σ_1' has prefix $\sigma_2'\sigma_3'a$ such that $\sigma_2'\sigma_3'a \in \mathcal{L}(\sigma_2'\sigma_O'\sigma\sigma_I'a)$ for some $a \in O \setminus O_\sigma$. By definition, since σ_I' contains only inputs and a is an output, we have that $\mathcal{L}(\sigma_2'\sigma_O'\sigma\sigma_I'a) \subseteq \mathcal{L}(\sigma_2'\sigma_O'\sigma a\sigma_I')$. Therefore σ_1' has prefix $\sigma_2'\sigma_3'a$ such that $\sigma_2'\sigma_3'a \in \mathcal{L}(\sigma_2'\sigma_O'\sigma a\sigma_I')$ for some $a \in O \setminus O_\sigma$. Since $\sigma_2'\sigma_O'\sigma a\sigma_I'$ does not satisfy property P, we can set $\sigma_1' = \sigma_2'\sigma_O'\sigma a\sigma_I'$ and the result follows.

By Proposition 3 we know that $\mathcal{A}(\sigma, O_\sigma)$ has $O(|\sigma|^2)$ states. In addition, we can construct the relation $\ll$ and the set of anti-chains in $O(|\sigma|^2)$ time. The following is therefore clear.

Proposition 5. *Given property (σ, O_σ), the process of generating the automata $\mathcal{A}(\sigma, O_\sigma)$ takes $O(|\sigma|^2)$ time.*

In passive testing we can update the current state of $\mathcal{A}(\sigma, O_\sigma)$ whenever we make a new observation and thus the complexity of applying passive testing is linear in the length of the trace that the SUT is producing. The following shows that the process is polynomial in the length of σ, which suggests that it can be applied in real-time since properties, and so $|\sigma|$, are usually relatively small.

Proposition 6. *Given property (σ, O_σ), the process of updating the state of $\mathcal{A}(\sigma, O_\sigma)$ takes $O(|\sigma|^4)$ time when a new input or output is observed.*

Proof. At each point in the process of simulating $\mathcal{A}(\sigma, O_\sigma)$ with a trace we have a current set of states. Consider that a new action a is observed and the set of states before this is S'. We know that $\mathcal{A}(\sigma, O_\sigma)$ has $O(|\sigma|^2)$ states. Assume that we have a list of transitions sorted by label. Using a binary search we can locate the start of the transitions with label a in time that is of order of the logarithm of the number of transitions and so in $O(\log |\sigma|)$ time. We can then determine the set of states after a by including those that are reached from states in S' by transitions with label a. We can find the transitions that leave states in S' in $O(|\sigma|^2)$ time and for each such state we can determine in $O(|\sigma|^2)$ time which states can be reached by a. Thus, the overall time complexity is of $O(|\sigma|^4)$ time.

5 Conclusions and Future Work

Testing is widely used to increase the confidence regarding the correctness of a system. Testing activities can be *formalised* when the tester has a formal definition of the properties that the system must have. Testing is usually interactive: the tester provides inputs to the SUT, receives outputs, and determines whether the outputs match the expected result. However, in some situations the tester cannot interact with the SUT and testing becomes passive. In this case, passive testing techniques replace classical (active) testing ones. In this paper we studied a formal methodology to passively test systems where communication is asynchronous. This is a topic that, as far as we know, has received little attention in the formal passive testing community.

In order to use our methodology in real-time, increasing performance and decreasing the needs for storage, we transform properties into automata that *approximately* capture the original property. We proved that in general an asynchronous framework it is not possible to construct a finite automata that accepts only the traces matching the original property. In order to construct our automata we used a classical mathematical structure, called *ideals*, so that the observation of an action changes the current state of the automaton from one of the associated ideals to another. We proved that the main operations, updating automata and checking traces, can be done in polynomial time, reinforcing the suitability of our methodology for use in real-time. We also defined notions of soundness and precision. The automaton A is sound for property P if whenever the SUT produces a trace that does not satisfy P, the observation made leads the automaton A to a final state (indicating a failure). The automaton A is precise for property P if whenever the observation made leads the automaton A to a final state (indicating a failure) we must have that the observation was one that could be made by the SUT failing to satisfy P. It transpired that our approach returned an automaton that is sound and precise.

This paper is the first step towards a complete theory of formal passive testing of systems with asynchronous communication. A first line to continue our work consists in drawing stronger conclusions from our observations. As we previously said, we are working on two lines. The first one considers different classes of properties. We have started to define simple properties that refer only to sequences of inputs or only to sequences of outputs and ignore the other observations. For example, "if we see $!o$ then the next output must be $!o'$." Other types of properties, inspired by classical work on temporal logics, are "a sequence eventually happens", "a sequence never happens" and "if a sequence happens, then we shouldn't observe a certain action".

A second line of work considers the inclusion of time information taking as initial step our work on testing in the distributed architecture with time information [17]. As discussed above, in general we cannot be certain that a particular sequence σ was produced by the SUT even if we observe a trace in $\mathcal{L}(\sigma)$. However, suppose that we know that there can be at most time t_m between an output being produced and it being observed and between an input being observed and it being received by the SUT. Further, suppose that if the SUT is in a state where it can produce output (possibly after a sequence of internal transitions) and it does so then this is within time t_o. Then, if time $2t_m + t_o$ passes without any observations being made then we know that the SUT must have reached a *quiescent* state: one from which it cannot produce output without first receiving input. If quiescence is observable then sometimes we can know that a property (σ, O_σ) has not been satisfied. There is additional scope to strengthen the conclusions that can be made. For example, if we have property $P = (\sigma, O_\sigma)$ where σ is an input sequence and we observe $\sigma\sigma'!o$ for input sequence σ' and output $!o \notin O_\sigma$ then we can conclude that the SUT failed P if there is a sufficient gap between the sending of the last input in σ and the output $!o$ being observed.

A third line of work considers more sophisticated mechanisms to improve the framework. First, we could have probabilities associated with *swapping* the order of actions and reasoning about how likely it is that an observation resulted from a property failing. Another improvement would be to consider stochastic information regarding delays. That is, probability distributions, rather than fix bounds, would be used to decide when we should expect that an action will not be observed in the future.

Finally, we would like to create a tool to support our theory and analyse realistic use cases to assess the applicability and usefulness of our framework.

Acknowledgements. We would like to thank the reviewers of the paper for the careful reading.

References

1. Andrés, C., Merayo, M.G., Núñez, M.: Formal passive testing of timed systems: Theory and tools. Software Testing, Verification and Reliability 22(6), 365–405 (2012)
2. Bayse, E., Cavalli, A., Núñez, M., Zaïdi, F.: A passive testing approach based on invariants: Application to the WAP. Computer Networks 48(2), 247–266 (2005)
3. Berry, G., Sethi, R.: From regular expressions to deterministic automata. Theoretical Computer Science 48(3), 117–126 (1986)
4. Cavalli, A., Gervy, C., Prokopenko, S.: New approaches for passive testing using an extended finite state machine specification. Information and Software Technology 45(12), 837–852 (2003)
5. Colombo, C., Pace, G.J., Abela, P.: Safer asynchronous runtime monitoring using compensations. Formal Methods in System Design 41(3), 269–294 (2012)
6. Dilworth, R.P.: A decomposition theorem for partially ordered sets. Annals of Mathematics 51(1), 161–166 (1950)
7. Gaudel, M.-C.: Testing can be formal, too! In: Mosses, P.D., Nielsen, M. (eds.) TAPSOFT 1995. LNCS, vol. 915, pp. 82–96. Springer, Heidelberg (1995)
8. Grieskamp, W., Kicillof, N., Stobie, K., Braberman, V.: Model-based quality assurance of protocol documentation: tools and methodology. Software Testing, Verification and Reliability 21(1), 55–71 (2011)
9. Hagenah, C., Muscholl, A.: Computing epsilon-free NFA from regular expressions in $O(n \cdot \log(n)^2)$ time. Informatique Théorique et Applications 34(4), 257–278 (2000)

10. Hennie, F.C.: Fault-detecting experiments for sequential circuits. In: 5th Annual Symposium on Switching Circuit Theory and Logical Design, pp. 95–110. IEEE Computer Society (1964)
11. Henniger, O.: On test case generation from asynchronously communicating state machines. In: 10th Int. Workshop on Testing of Communicating Systems, IWTCS 1997, pp. 255–271. Chapman & Hall (1997)
12. Hierons, R.M.: The complexity of asynchronous model based testing. Theoretical Computer Science 451, 70–82 (2012)
13. Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Luettgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., Zedan, H.: Using formal specifications to support testing. ACM Computing Surveys 41(2) (2009)
14. Hierons, R.M., Bowen, J.P., Harman, M. (eds.): Formal Methods and Testing. LNCS, vol. 4949. Springer, Heidelberg (2008)
15. Hierons, R.M., Merayo, M.G., Núñez, M.: Scenarios-based testing of systems with distributed ports. Software - Practice and Experience 41(10), 999–1026 (2011)
16. Hierons, R.M., Merayo, M.G., Núñez, M.: Implementation relations and test generation for systems with distributed interfaces. Distributed Computing 25(1), 35–62 (2012)
17. Hierons, R.M., Merayo, M.G., Núñez, M.: Using time to add order to distributed testing. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 232–246. Springer, Heidelberg (2012)
18. Hromkovic, J., Seibert, S., Wilke, T.: Translating regular expressions into small ϵ-free nondeterministic finite automata. Journal of Computer Systems and Science 62(4), 565–588 (2001)
19. Huo, J.L., Petrenko, A.: On testing partially specified IOTS through lossless queues. In: Groz, R., Hierons, R.M. (eds.) TestCom 2004. LNCS, vol. 2978, pp. 76–94. Springer, Heidelberg (2004)
20. Huo, J., Petrenko, A.: Transition covering tests for systems with queues. Software Testing, Verification and Reliability 19(1), 55–83 (2009)
21. Lee, D., Netravali, A.N., Sabnani, K.K., Sugla, B., John, A.: Passive testing and applications to network management. In: 5th IEEE Int. Conf. on Network Protocols, ICNP 1997, pp. 113–122. IEEE Computer Society (1997)
22. Mammar, A., Cavalli, A., Jimenez, W., Mallouli, W., Montes de Oca, E.: Using testing techniques for vulnerability detection in C programs. In: Wolff, B., Zaïdi, F. (eds.) ICTSS 2011. LNCS, vol. 7019, pp. 80–96. Springer, Heidelberg (2011)
23. Mealy, G.H.: A method for synthesizing sequential circuits. Bell System Techical Journal 34, 1045–1079 (1955)
24. Morales, G., Maag, S., Cavalli, A.R., Mallouli, W., Montes de Oca, E., Wehbi, B.: Timed extended invariants for the passive testing of web services. In: 8th IEEE Int. Conf. on Web Services, ICWS 2010, pp. 592–599. IEEE Computer Society (2010)
25. Simão, A., Petrenko, A.: Generating asynchronous test cases from test purposes. Information and Software Technology 53(11), 1252–1262 (2011)

Input-Output Conformance Simulation (iocos) for Model Based Testing[*]

Carlos Gregorio-Rodríguez, Luis Llana, and Rafael Martínez-Torres

Departamento Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
cgr@sip.ucm.es, llana@ucm.es, rmartine@fdi.ucm.es

Abstract. A new model based testing theory built on simulation semantics is presented. At the core of this theory there is an **i**nput-**o**utput **co**nformance **s**imulation relation (iocos). As a branching semantics iocos can naturally distinguish the context of local choices. We show iocos to be a finer relation than the classic ioco conformance relation. It turns out that iocos is a transitive relation and therefore it can be used both as a conformance relation and a refinement preorder. An alternative characterisation of iocos is provided in terms of testing semantics. Finally we present an algorithm that produces a test suite for any specification. The resulting test suite is sound and exhaustive for the given specification with respect to iocos.

Keywords: Model Based Testing, Input Output Conformance Simulation, Formal Methods.

1 Introduction and Related Work

Model-Based Testing (MBT) is an active research area whose goals are to increase correctness and efficiency of the testing process and, at the same time, reducing the costs, in a world of ever increasingly complex systems. From the quite ad-hoc techniques used in seminal papers (see for instance [5]), the use of formal methods in MBT and, in particular, behavioural models to describe the system specification, has made it possible to develop theories, frameworks and tools that automatically produce efficient set of test cases from a model. This allows an effective automation of the testing process to assess whether the system under test behaves as expected by its specification.

A key point at the core of any MBT theory is the implementation or conformance relation, stating whether an implementation is *correct*, in some sense, with respect to a given specification. Tretmans' input-output conformance relation (ioco) [19] is one of the most established ones. A whole MBT framework has been developed around the ioco relation, from theory to tools [20,21,4].

[*] Research partially supported by the Spanish MEC projects TIN2009-14312-C02-01 and TIN2012-36812-C02-01.

As a behavioural relation over labelled transition systems, ioco can be classified as a linear semantics [9], as most of the relations used in MBT are. That means that they are essentially based on traces and that brings advantages and drawbacks well known in process theory.

One of the disadvantages of linear semantics is the limitation to observe the execution context, that is, the different available choices at a given point. The simple behaviours in Figure 1 highlight this situation. [1] One of the goals guiding our research is precisely to find a MBT theory capable of identifying the implementations conforming not only the trace executions, but also the choices in the specification.

The starting point for our work has been inspired in basic results in process theory: branching semantics [9] form a family of relations that can naturally distinguish the execution context for a process. These semantics are essentially based on simulation [13], they are easily defined by coinduction and thus coalgebraic techniques can be applied in its study. Simulation can be characterised as a game, there exist algorithms to compute it [16,10] and, together with bisimulation [14], is a rather natural and pervasive concept that appears in many different contexts in the literature.

However, there is not much work on MBT and simulation relations. As far as we know the only related work is the one developed by a group at *Microsoft Research* that has several publications on MBT (see for instance [24,23]). Their framework is built on the alternating simulation relation [3,2] defined for interface automata.

This paper is organized as follows: in Section 2, we introduce the formal framework used to model behaviours and some technical definitions and notations used along the paper, including the classic ioco relation. Section 3 presents a reasoned exposition of examples discussing what might and might not be considered a correct implementation of a given specification. This exposition highlights differences and similarities between conformance relations based on linear and branching semantics. These examples settle also the goals that we want to fulfil with the formal definition of the iocos relation. We finish this section by proving iocos to be a refinement of ioco on input-output labelled transition systems. Section 4 focuses on describing and proving the results that show iocos to be a suitable relation for MBT. First, we provide a language for tests and formally define the test execution of a behaviour or implementation. Then we show that iocos can be characterised with a preorder defined in the classic style of testing semantics [15,11]. Finally, we define an algorithm to automatically generate a test suite from a given specification, we prove that the implementations that pass this test suite are exactly those that are in iocos relation with the given specification. Lastly, Section 5 summarises the goals achieved as well as some future lines of research.

[1] In Section 3 we thoroughly get into the details and use further examples to illustrate the limitations of linear semantics.

2 Preliminaries

A common formalism used in MBT to represent not only the models but also the implementations and even the tests are labelled transition systems. In order to deal with input-output behaviours we are going to consider two disjoint finite sets of actions: inputs I and outputs O. Output actions are those initiated by the system, they will be annotated with an exclamation mark, $a!, b!, x!, y! \in O$. Input actions are initiated by the environment and will be annotated with a question mark, $a?, b?, x?, y? \in I$. In many cases we want to name actions in a general sense, inputs and outputs indistinctly. We will consider the set $L = I \cup O$ and we will omit the exclamation or question marks when naming generic actions, $a, b, x, y \in L$.

A state with no output actions cannot autonomously proceed, such a state is called *quiescent*. Quiescence is an essential component of the ioco theory. For the sake of simplicity and without lost of generality (see for instance [20,18]), we directly introduce the event of quiescence as a special action denoted by δ into the definition of our models.

Definition 1. A *labelled transition system with inputs and outputs* is a 4-tuple $(S, I, O, \rightarrow)$ such that

- S is a set of states or behaviours.
- I and O are disjoint sets of input and output actions respectively.We define $L = I \cup O$ and consider a new symbol $\delta \notin L$ for *quiescence*. We will consider also the sets $L_\delta = L \cup \{\delta\}$ and $O_\delta = O \cup \{\delta\}$.
- $\rightarrow \subseteq S \times L_\delta \times S$. As usual we write $p \xrightarrow{a} q$ instead of $(p, a, q) \in \rightarrow$ and $p \xrightarrow{a}$, for $a \in L_\delta$, if there exists $q \in S$ such that $p \xrightarrow{a} q$. Analogously, we will write $p \overset{a}{\not\rightarrow}$, for $a \in L_\delta$, if there is no q such that $p \xrightarrow{a} q$. In order to allow only coherent quiescent systems the set of transitions should also satisfy:
  - if $p \xrightarrow{\delta} p'$ then $p = p'$. A quiescent transition is always reflexive.
  - if $p \overset{o!}{\not\rightarrow}$ for any $o! \in O$, then $p \xrightarrow{\delta} p$. A state with no outputs is quiescent.
  - if there is $o! \in O$ such that $p \xrightarrow{o!}$, then $p \overset{\delta}{\not\rightarrow}$. A quiescent state performs no output actions.

□

For the sake of simplicity, we will denote the set of labelled transition systems with inputs and outputs just as *LTS*. In general we use $p, q, p', q' \ldots$ for states or behaviours, but also i, i', s and s' when we want to emphasise the concrete role of a behaviours as implementation or specification.

Without losing generality, we will consider implementations and specifications, or, more in general, behaviours under study, as states of the same LTS[2]. This modification simplifies the coinductive definition we are going to present and the reasoning in the proofs.

[2] If we had two different *LTSs*, one for a specification and one for the implementation, we could always consider the larger *LTS* that is the disjoint union of the original *LTSs*.

Traces play an important role gathering basic information for behaviours. A trace is a finite sequence of symbols of L_δ. We will normally use the symbol σ to denote traces, that is, $\sigma \in L_\delta^*$. The empty trace is denoted by ϵ and we juxtapose, $\sigma_1\sigma_2$, to indicate concatenation of traces. The transition relation of labelled transition systems can naturally be extend using traces instead of single actions.

Definition 2. Let $(S, I, O, \rightarrow) \in LTS$, $p, q \in S$ and $\sigma \in L_\delta^*$. We inductively define $p \xrightarrow{\sigma} q$ as follows:

– $p \xrightarrow{\epsilon} p$
– $p \xrightarrow{a\sigma} q$ for $a \in L_\delta$, $\sigma \in L_\delta^*$ and $p' \in S$ such that $p \xrightarrow{a} p'$ and $p' \xrightarrow{\sigma} q$. □

Next we introduce some definitions and notation that will be frequently used along the paper.

Definition 3. Let $(S, I, O, \rightarrow) \in LTS$, and $p \in S$, $S' \subseteq S$, and $\sigma \in L_\delta^*$, we define:

1. $\mathsf{init}(p) = \{a \mid a \in L_\delta,\ p \xrightarrow{a}\}$, the set of initial actions of p.
2. $\mathsf{traces}(p) = \{\sigma \mid \sigma \in L_\delta^*,\ p \xrightarrow{\sigma}\}$, the set of traces from p.
3. $p\ \mathsf{after}\ \sigma = \{p' \mid p' \in S,\ p \xrightarrow{\sigma} p'\}$, the set of reachable states from p after the execution of trace σ.
4. $\mathsf{outs}(p) = \{x \mid x \in O_\delta,\ p \xrightarrow{x}\}$, the set of outputs of a state p or the quiescent symbol δ.
5. $\mathsf{outs}(S') = \bigcup_{p \in S'} \mathsf{outs}(p)$, the set of outputs of a set of states S'.
6. $\mathsf{ins}(p) = \{x? \mid x? \in I,\ p \xrightarrow{x?}\}$, the set of inputs of a state p. □

A classical requirement for the ioco relation in [20] is that implementations should be *input enabled*, that means that the system is always prepared to perform any input action and therefore all inputs are enabled in all states. Although this assumption maybe natural in some contexts is not so in others. For instance, in a vending machine, a slot, for a credit card or parking ticket, can be only enabled if a card is not inserted; much alike, developers of graphical interfaces do not need to consider any possible event on a window, they just code the response for the interesting events, etc.

Moreover, in the ioco theory, while implementations must be input enabled, specifications do not need to fulfil this requirement. So the original ioco relation is defined between two different domains, general input-output labelled transition systems for specifications and input enabled input-output labelled transition systems for implementations. Hence, the ioco relation is not transitive and cannot be used as a refinement relation: once an implementation conforms a specification, the implementation fixes all the behaviour regarding the input actions; so there is little freedom, if any, to continue the refining process.

In our framework we do not require the implementations to be input enabled. As usual in other testing frameworks, specifications and implementations are expressed in the same formalism, in particular we are going to use *LTSs* for both implementations and specifications.

In order to compare the original ioco relation with the conformance relation we are going to define in next section, we have to adapt the ioco definition to our framework.

Definition 4. Let $(S, I, O, \rightarrow) \in LTS$, the relation $\mathsf{ioco} \subseteq S \times S$ is defined as follows: $i \;\mathsf{ioco}\; s \Leftrightarrow_{def} \forall \sigma \in \mathsf{traces}(s) : \mathsf{outs}(i \;\mathsf{after}\; \sigma) \subseteq \mathsf{outs}(s \;\mathsf{after}\; \sigma)$ □

The ioco relation we use keeps the spirit of the original in [20], but while the original imposed implementations to be input enabled, our definition has been extended to the more general domain of input-output labelled transition systems. Also, the original definition used "suspension traces" (Definition 9 in [20]) while we can consider just traces because the quiescence symbol has already been introduced in the description of the behaviours.

3 Input-Output Conformance Simulation (IOCOS)

In this section we will present the alternative relation that we propose. We have defined this relation according to the following criteria in mind: first, the ioco relation is a well known and accepted relation, we want to find a refinement of the original ioco relation while keeping as close as possible to it. Second, it should be defined as a simulation relation, so we can benefit from the work in this field. Finally, there should be a testing framework, similar to the ones in [1,20]. In order to simplify the reading, we are going to mark the quiescent states as $\circ$, these kind of nodes are shorthands for $\bullet\!\circlearrowleft^{\delta}$.

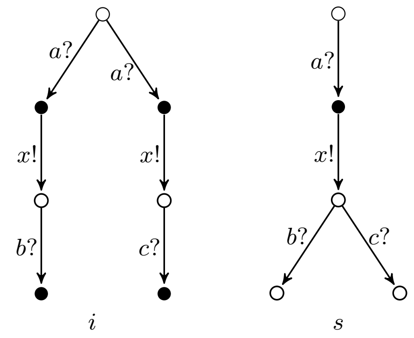


Fig. 1. i ioco s and $i \not\mathsf{iocos}\; s$

Next we are going to present some examples that motivate the definition of iocos (Definition 5). The rationale behind the ioco definition is to serve as an *observation methodology* to relate a specification and an implementation. This methodology binds the environment to *traces*, the observations to *output signals*, and comparison to *set inclusion*. Taking traces as environments prevent to distinguish non-determinism. Looking at Figure 1, there are arguments to discard i as a sound implementation of s: after the trace $a?x!$, s can react *always* to both signals $b?$ and $c?$, while this is not true for i: depending on the selected branch, i can only react to either $b?$ or $c?$ but not both. As we will see, simulation tecniques provide us with necessary insight to solve this problem.

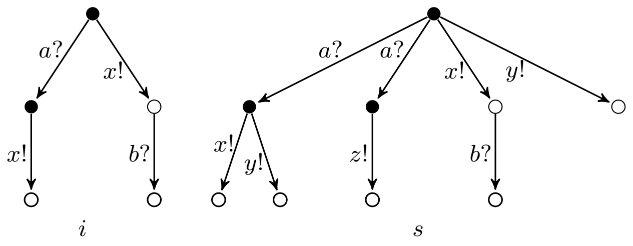


Fig. 2. i ioco s and i iocos s

There are two distinguised trends when dealing with specifications: the *initial* semantics, where a specification sets a minimum that an implementation must fit to be qualified as sound; and the *final* semantics, where specification stands rather like a limit for implementation's behaviour. Both are legitimate, but subset relation $\subseteq$ at Definition 4, unveils ioco clearly in this last category. We also adopt this finality approach: any behaviour of a correct implementation must be considered valid by the specification. Graphically, Figure 2, can summarize this with the next idea: any subtree in the implementation is a subtree in the specification.

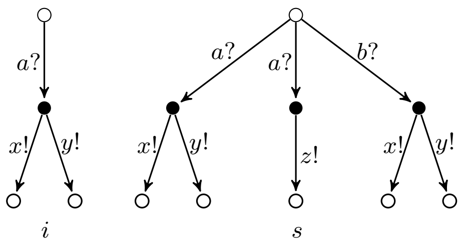


Fig. 3. i ioco s and i $\cancel{\text{iocos}}$ s

Traditionally, an objection is made to this approach: partial or even empty implementations can be accepted as sound. In this sense we want to reach a trade-off between the previous postulate, specification as limit, while avoid such tricky implementations.

This subttle requirement can be formulated for input actions in the following terms: at least one of each of input actions considered by the specification should be implemented. In this way we can have arguments to discard i as a sound implementation of s in Figure 3: the implementation cannot react to the input action $b?$. However, limits on input actions should be applicable only to those prompted by the specification. Beyond that, implementation should be free to behave. This is the case of the input action $b?$ in Figure 4.

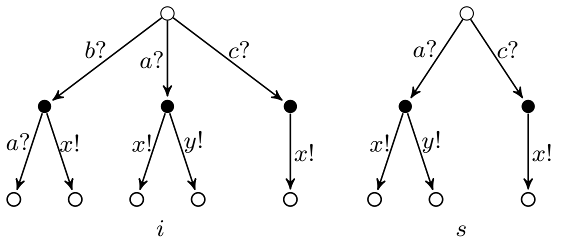


Fig. 4. i ioco s and i iocos s

Now we can give the formal definition of iocos. It reflects the ideas presented in the examples above. Since it is a simulation relation, it cannot be defined directly. So first we give the notion of an iocos-relation. Then the iocos relation would be the union of all iocos-relations.

Definition 5. Let $(S, I, O, \rightarrow) \in LTS$, we say that a relation $R \subseteq S \times S$ is a iocos-relation iff for any $(p, q) \in R$ the following conditions hold

1. $\mathsf{ins}(q) \subseteq \mathsf{ins}(p)$
2. For all $a? \in \mathsf{ins}(q)$ such that $p \xrightarrow{a?} p'$ there exists $q' \in S$ such that $q \xrightarrow{a?} q'$ and $(p', q') \in R$. [3]
3. For all $x \in \mathsf{outs}(p)$ such that $p \xrightarrow{x} p'$ there exists $q' \in S$ such that $q \xrightarrow{x} q'$ and $(p', q') \in R$.

We define the *input-output conformance simulation* as

$$\mathsf{iocos} = \bigcup \{R \mid R \subseteq S \times S,\ R \text{ is a } \mathsf{iocos}\text{-relation}\}$$

and we write $p\,\mathsf{iocos}\,q$ instead of $(p, q) \in \mathsf{iocos}$. □

Next technical results show that iocos is indeed a well defined preorder relation on LTS, for this purpose we need the following lemma.

Lemma 1. Let $(S, I, O, \rightarrow) \in LTS$, the following properties hold:

– $Id = S \times S$ is a iocos-relation.
– Let $R, R' \subseteq S \times S$ be two iocos-relations, then $R \circ R' = \{(p, r) \mid \exists q \in S : (p, q) \in R \wedge (q, r) \in R'\}$ is a iocos-relation.

□

Corollary 1. Let $(S, I, O, \rightarrow) \in LTS$, then iocos is a preorder. □

The rest of this section is devoted to prove that iocos is a finer relation than ioco. Let us start proving a simple lemma that is a direct consequence of the definition of the function after.

[3] Let us note that the Condition 2 does not imply Condition 1.

Lemma 2. Let $(S, I, O, \rightarrow) \in LTS$, $p \in S$, $a \in L_\delta$ and $\sigma' \in L_\delta^*$, then

$$p \,\mathsf{after}\, a\sigma' = \bigcup \{p' \,\mathsf{after}\, \sigma' | p \xrightarrow{a} p'\}$$

□

Theorem 1. Let $(S, I, O, \rightarrow) \in LTS$ then $\mathsf{iocos} \subseteq \mathsf{ioco}$. That is, for any $p, q \in S$, whenever we have $p \,\mathsf{iocos}\, q$ it is also true that $p \,\mathsf{ioco}\, q$.

Proof. Let us consider $p, q \in S$ such that $p \,\mathsf{iocos}\, q$. According to Definition 4, it suffices to prove

$$\forall \sigma \in \mathsf{traces}(q) : \mathsf{outs}(p \,\mathsf{after}\, \sigma) \subseteq \mathsf{outs}(q \,\mathsf{after}\, \sigma).$$

We will prove this by induction on length of trace σ.

Basis: $\sigma = \epsilon$**.** First, $\mathsf{outs}(p \,\mathsf{after}\, \epsilon) = \mathsf{outs}(p)$ and $\mathsf{outs}(q \,\mathsf{after}\, \epsilon) = \mathsf{outs}(q)$, by definition of the after function. Then $\mathsf{outs}(p) \subseteq \mathsf{outs}(q)$, by Definition 5.3 of iocos.

Induction: $\sigma = a\sigma', a \in L_\delta$. If $p \overset{a}{\not\rightarrow}$ then $\sigma \notin \mathsf{traces}(p)$ and $\mathsf{outs}(p \mathsf{after} \sigma) = \varnothing$. So let us consider p' such that $p \xrightarrow{a} p'$. Since $\sigma \in \mathsf{traces}(q)$, $q \xrightarrow{a}$. If $a \in I$ then $a \in \mathsf{ins}(q)$ and by Definition 5.2 there is $q' \in S$ such that $q \xrightarrow{a} q'$ and $p' \,\mathsf{iocos}\, q'$. If $a \in O_\delta$, then by Definition 5.3 there is $q' \in S$ such that $q \xrightarrow{a} q'$. and $p' \,\mathsf{iocos}\, q'$. Hence, in any case there is $q' \in S$ such that $q \xrightarrow{a} q'$ and $p' \,\mathsf{iocos}\, q'$. By induction $\mathsf{outs}(p' \,\mathsf{after}\, \sigma') \subseteq \mathsf{outs}(q' \,\mathsf{after}\, \sigma')$. Therefore:

$$\mathsf{outs}(p \,\mathsf{after}\, \sigma) = \mathsf{outs}(p \,\mathsf{after}\, a\sigma') = \bigcup \{\mathsf{outs}(p' \,\mathsf{after}\, \sigma') \mid p \xrightarrow{a} p'\} \subseteq$$

$$\bigcup \{\mathsf{outs}(q' \,\mathsf{after}\, \sigma') \mid p \xrightarrow{a} p', q \xrightarrow{a} q', p' \,\mathsf{iocos}\, q'\} \subseteq$$

$$\bigcup \{\mathsf{outs}(q' \,\mathsf{after}\, \sigma') \mid q \xrightarrow{a} q'\} = \mathsf{outs}(q \,\mathsf{after}\, a\sigma') = \mathsf{outs}(q \,\mathsf{after}\, \sigma)$$ □

Complementing the previous result, examples in Figures 1 and 3 show that iocos is a strict refinement of ioco.

4 Testing Framework

After presenting the ideas behind the iocos relation and its essential basic properties, in this section, we show that iocos can indeed be a suitable relation for MBT: for any specification a test suite can be automatically derived such that an implementation would be correct (wrt iocos) if and only if it passes all the tests in the test suite.

The technical approach we use to achieve these results is slightly different to the one used by Tretmans for ioco. We first show iocos to have a characterisation as a testing semantics, Definition 9, in the classic sense of De Nicola and Hennessy. Then we define an algorithm, Definition 10, that for a given behaviour generates a set of tests (test suite) that can discriminate the suitable implementations or refinements of that behaviour, Theorem 3. We use the testing characterisation to prove this result.

4.1 Tests Definition and Execution

The kind of experiments we have to conduct on behaviours are called tests. Tests would play the role of environments for the implementations.

There are two kind of choices in the tests: the one corresponding to the $+$ operator and the one corresponding to the $\oplus$ operator. The former is the usual choice operator as in [20]. The latter, borrowed from [1], corresponds to an *or* operator: in order p to pass $T_1 \oplus T_2$ it is enough that P passes either T_1 or T_2. As in [1], the presence of these choice operators implies the ability of make copies of the machine at intermediate points. Then it is necessary to perform the tests on the copies, and finally to combine the results to obtain the outcome of the overall test.

Definition 6. A *test* is a syntactical term defined by the following Backus-Naur Form:

$$T = \text{✗} \mid \checkmark \mid T_1 \oplus T_2 \mid T_1 + T_2 \mid a;T \qquad \text{where } a \in L_\delta$$

We denote the set of tests as $\mathcal{T}$. □

As usual in MBT, the environments we want to model with tests have some added particularities that we need to consider. First, tests should be able to respond at any moment to any possible output of the implementation under test. That is, tests like $a?;T_{a?}$ with $a? \in I$, will not be accepted as valid tests. These test should be completed into tests like $a?;T_{a?} + \sum_{x\in O_\delta} x;T_x$. For the sake of simplicity we use $\sum_{i\in\{1,\ldots,n\}} T_i$ as a shortcut for $T_1 + \cdots + T_n$. Analogously, $o!;T_{o!}$ is not an acceptable test, instead we need to consider the test $o!;T_{o!} + \sum_{x\in O_\delta,\ x\neq o!} x;T_x$. All these conditions are reflected in the following definition.

Definition 7. Let $T \in \mathcal{T}$ be a test, T is *valid* iff it has one of the following forms:

1. $T = \text{✗}$ or $T = \checkmark$.
2. $T = a?;T_{a?} + \sum_{x\in O_\delta} x;T_x$ where $x \in O_\delta$, $a? \in I$, and $T_{a?}$, T_x are valid tests.
3. $T = \sum_{x\in O_\delta} x;T_x$ where T_x is a valid test for $x \in O_\delta$.
4. $T = T_1 \oplus T_2$ where T_1 and T_2 are valid tests.

We denote the set of valid tests as $\mathcal{T}_v$. □

So far we have a language for tests, and we have now to define how these tests interacts with a behaviour and what will the result of the execution of that experiment be. Following the ideas of Abramsky in [1] we use a predicate to define the outcomes of the interaction between a test and the behaviour or implementation being tested.

Definition 8. Let $(S, I, O, \rightarrow) \in LTS$, we inductively define the predicate $\mathsf{pass} \subseteq S \times \mathcal{T}$ as follows (let us assume that $s \in S$, $a? \in I$, and $x \in O_\delta$)

$$
\begin{aligned}
&s \;\mathsf{pass}\; ✗ = false\\
&s \;\mathsf{pass}\; ✓ = true\\
&s \;\mathsf{pass}\; x; T_x = \begin{cases} true & \text{if } x \notin \mathsf{outs}(s)\\ \bigwedge\{s' \;\mathsf{pass}\; T_x | s \xrightarrow{x} s'\} & \text{otherwise}\end{cases}\\
&s \;\mathsf{pass}\; a?; T_{a?} = \begin{cases} false & \text{if } a? \notin \mathsf{ins}(s)\\ \bigwedge\{s' \; pass \; T_{a?} | s \xrightarrow{a?} s'\} & \text{otherwise}\end{cases}\\
&s \;\mathsf{pass}\; T_1 + T_2 = s \;\mathsf{pass}\; T_1 \wedge s \;\mathsf{pass}\; T_2\\
&s \;\mathsf{pass}\; T_1 \oplus T_2 = s \;\mathsf{pass}\; T_1 \vee s \;\mathsf{pass}\; T_2
\end{aligned}
$$

□

Let us note that for the sake of convenience the predicate pass is defined over the whole set of tests, with a simpler structural formulation, while at the end we will only be interested in valid tests.

Next let us show the tests that discriminate i and s in the previous examples when $i \,\not\!\mathsf{iocos}\, s$. The test $T = a?; x!; b?; ✓$ differentiates the behaviours in Figure 1. It is passed by the specification s but not by the implementation i. Let us note that this test is not valid. But its related valid test

$$T^v = a?; (x!; (b?; ✓ + x!; ✗ + \delta; ✓) + \delta; ✗) + x!; ✗ + \delta; ✓$$

also differenciates i and s.

The next test is related to Figure 2. Since $i \,\mathsf{iocos}\, s$, because of Theorem 2, we cannot find a test that distinguishes i and s. Since the number of potential test is infinite, it is not feasible to check all of them. In this case there is a *maximal test* according to the algorithm in Definition 10. This test is the following one:

$$
\begin{aligned}
&let\\
&\quad T_{a_1?} = (x!✓ + y!; ✓ + z!; ✗ + \delta; ✗)\\
&\quad T_{a_2?} = (z!; ✓ + x!; ✗ + y!; ✗ + \delta; ✗)\\
&\quad T_{b?} = (b?✓ + x!; ✗ + y!; ✗ + z!; ✗ + \delta; ✓)\\
&in\\
&\quad T = a?; (T_{a_1?} \oplus T_{a_2?}) + x!; T_{b?} + y!; ✓ + z!; ✗ + \delta; ✗
\end{aligned}
$$

It is easy to check that both $i \,\mathsf{pass}\, T$ and $s \,\mathsf{pass}\, T$. All other tests generated according to Defintion 10 can be built from the previous test by pruning branches.

For the specification s in Figure 3 there are two maximal tests according to Defintion 10:

$$
\begin{aligned}
&let\\
&\quad T_{a_1?} = (x!✓ + y!; ✓ + z!; ✗ + \delta; ✗)\\
&\quad T_{a_2?} = (z!; ✓ + x!; ✗ + y!; ✗ + \delta; ✗)\\
&\quad T_{b?} = (x!; ✓ + y!; ✓ + z!; ✗ + \delta; ✗)\\
&in\\
&\quad T_1 = a?; (T_{a_1?} \oplus T_{a_2?}) + x!; ✗ + y!; ✗ + z!; ✗ + \delta; ✓\\
&\quad T_2 = b?; T_{b?} + x!; ✗ + y!; ✗ + z!; ✗ + \delta; ✓
\end{aligned}
$$

It is easy to check that T_1 is passed by s and i, but T_2 is only passed by s.

4.2 Testing Characterisation of iocos

Now we are going to prove that the iocos relation can be characterise in terms of testing. With the pass predicate we have defined a notion of test execution. Upon this notion it is easy to define a testing preorder ($\sqsubseteq_T$) in terms of how many tests are passed: a behaviour will be *better* than other if the former passes more tests than the latter. This section is devoted to prove that this testing preorder is precisely the inverse of the iocos relation: $\mathsf{iocos} = {\sqsubseteq_T}^{-1}$, (Theorem 2).

Definition 9. Let $(S, I, O, \rightarrow) \in LTS$ and $p, q \in S$, we define the preorder

$$p \sqsubseteq_T q \text{ iff } \forall T \in \mathcal{T}_v :\ p \,\mathsf{pass}\, T \implies q \,\mathsf{pass}\, T$$

□

To improve the readability, the proof of Theorem 2 ($\mathsf{iocos} = {\sqsubseteq_T}^{-1}$) has been split into Proposition 1 (${\sqsubseteq_T}^{-1} \subseteq \mathsf{iocos}$) and Proposition 2 ($\mathsf{iocos} \subseteq {\sqsubseteq_T}^{-1}$).

Proposition 1. Let $(S, I, O, \rightarrow)$ and $p, q \in S$, if $q \sqsubseteq_T p$ then $p \,\mathsf{iocos}\, q$.

Proof. In order to prove $p \,\mathsf{iocos}\, q$ we must find an iocos-relation R such that $(p, q) \in R$. Let us define

$$R = \{(p_1, p_2) \mid p_1, p_2 \in S,\ p_2 \sqsubseteq_T p_1\}$$

It is clear that $(p, q) \in R$. We have to prove that R is an iocos-relation. We are going to prove it by contradiction, that is, if there is $(p_1, p_2) \in R$ that does not satisfy one of the conditions of the definition of an ioco-relation (Definition 5), then $p_2 \not\sqsubseteq_T p_1$. So we must find a test $T \in \mathcal{T}_v$ such that $p_2 \,\mathsf{pass}\, T$ but $p_1 \,\cancel{\mathsf{pass}}\, T$. Let us distinguish the cases according to the condition that the pair (p_1, p_2) does not hold:

(p_1, p_2) *does not hold 1 in Definition 5.* So there is $a? \in \mathsf{ins}(p_2)$ such that $a? \notin \mathsf{ins}(p_1)$. Let us consider the test:

$$T = a?\checkmark + \sum_{x \in O_\delta} x; \checkmark$$

It is clear that $p_1 \,\cancel{\mathsf{pass}}\, T$ but $p_2 \,\mathsf{pass}\, T$.

(p_1, p_2) *does not hold 2 in Definition 5.* We can assume that Definition 5.1 holds. Then, there is $a? \in \mathsf{ins}(p_1)$ and $p_1' \in S$ such that $p_1 \xrightarrow{a?} p_1'$ and $(p_1', p_2') \notin R$ for any p_2' such that $p_2 \xrightarrow{a?} p_2'$. Let us consider the set $P = \{p_2' \mid p_2 \xrightarrow{a?} p_2',\ (p_1', p_2') \notin R\}$. Since Definition5.1 holds, $P \neq \varnothing$. For any $r \in P$ there is a test T_r such that $r \,\mathsf{pass}\, T$ and $p_1' \,\cancel{\mathsf{pass}}\, T_r$. Then let us consider the test

$$T = a?; \bigoplus_{r \in P} T_r + \sum_{x \in O_\delta} x; \checkmark$$

Then $p_2 \,\mathsf{pass}\, T$ but $p_1 \,\cancel{\mathsf{pass}}\, T$.

(p_1, p_2) *does not hold 3 in Definition 5.* There is $x \in \mathsf{outs}(p_1) \cup \{\delta\}$ such that $p_1 \xrightarrow{x} p_1'$ but $(p_1', p_2') \notin R$ for any p_2 such that $p_2 \xrightarrow{x} p_2'$. Let us consider the set $P = \{p_2' \mid p_2 \xrightarrow{x} p_2',\ (p_1', p_2') \notin R\}$. If $P = \varnothing$, let us consider the test

$$T = x; ✗ + \sum_{x \neq y, y \in O_\delta} y; ✓$$

Then $p_2\ \mathsf{pass}\ T$ and $p_1\ \cancel{\mathsf{pass}}\ T$.
So let us suppose $P \neq \varnothing$. Then for any $r \in P$ there is T_r such that $r\ \mathsf{pass}\ T$ and $p_1'\ \cancel{\mathsf{pass}}\ T_r$. So let us consider the test

$$T = x; \bigoplus_{r \in P} T_r + \sum_{x \neq y, y \in O_\delta} y; ✓$$

Then $p_2\ \mathsf{pass}\ T$ and $p_1\ \cancel{\mathsf{pass}}\ T$. □

Proposition 2. Le $(S, I, O, \rightarrow) \in LTS$ and $p, q \in S$. If $p\ \mathsf{iocos}\ q$ then $q \sqsubseteq_T p$.

Proof. Let us consider a test $T \in \mathcal{T}_v$ such that $q\ \mathsf{pass}\ T$. We have to prove that $p\ \mathsf{pass}\ T$. Let us prove by structural induction on T.

$T = ✓$ or $T = ✗$. If $T = ✗$ then $q\ \cancel{\mathsf{pass}}\ T$ that is a contradiction. The test ✓ is passed for any $p \in S$.

$T = T_1 \oplus T_2$. By definition of q pass T then either q pass T_1 or q pass T_2. Let us assume q pass T_1, the other case is symmetric. By induction p pass T_1, therefore $p\ \mathsf{pass}\ T_1 \oplus T_2 = T$.

$T = \sum_{x \in O_\delta} x; T_x$. Let us consider the set $O' = \{x \mid x \in O_\delta, \exists p' : p \xrightarrow{x} p'\}$; $O' \neq \varnothing$, by definition of LTS. In order to prove that $p\ \mathsf{pass}\ T$ we have to prove that $p_x \mathsf{pass} T_x$ for any $p_x \in S$ such that $p \xrightarrow{x} p_x$. So let us consider any of these p_x. By Definition 5.3, there is q_x such that $q \xrightarrow{x} q_x$ and $p_x\ \mathsf{iocos}\ q_x$. Since $q\ \mathsf{pass}\ T$, $q_x\ \mathsf{pass}\ T_x$. Then by induction, $p_x\ \mathsf{pass}\ T_x$. Since this is true for any $x \in O'$ and $O' \neq \varnothing$, $p\ \mathsf{pass}\ T$.

$T = \sum_{x \in O_\delta} x; T_x + a?T_{a?}$. First $a? \in \mathsf{ins}(q)$ since $q\ \mathsf{pass}\ T$. By Definition 5.1, $\mathsf{ins}(q) \subseteq \mathsf{ins}(p)$, therefore $a? \in \mathsf{ins}(p)$. Let us consider $p_{a?} \in S$ such that $p \xrightarrow{a?} p_{a?}$. By Definition 5.2, there is $q_{a?} \in S$ such that $q \xrightarrow{a?} q_{a?}$ and $p_{a?}\ \mathsf{iocos}\ q_{a?}$. Since $q\ \mathsf{pass}\ T$, $q_{a?}\ \mathsf{pass}\ T_{a?}$ and then, by induction, $p_a\ \mathsf{pass}\ T_{a?}$. Like in the previous case let us consider the set $O' = \{x \mid x \in O_\delta, \exists p' : p \xrightarrow{x} p'\}$. Reasoning like in the previous case we obtain that $p_x\ \mathsf{pass}\ T_x$ for any $p_x \in S$ and $x \in O'$ such that $p \xrightarrow{x} p_x$.
So, we obtain:
- $p \xrightarrow{a?}$.
- For any $p_{a?} \in S$ such that $p \xrightarrow{a?} p_{a?}$ we obtain $p_{a?}\ \mathsf{pass}\ T_{a?}$.
- For any $x \in O_\delta$ such that $p \xrightarrow{x} p_x$ we obtain $p_x\ \mathsf{pass}\ T_x$.

Therefore $p\ \mathsf{pass}\ T$. □

Theorem 2. ($\mathsf{iocos} = \sqsubseteq_T{}^{-1}$) Let $(S, I, O, \rightarrow) \in LTS$ and $i, s \in S$, then $i\ \mathsf{iocos}\ s$ iff $s \sqsubseteq_T i$. □

4.3 Test Generation

In Section 4.2 we have showed that $i \,\mathsf{iocos}\, s$ iff $\forall T \in \mathcal{T}_v : \; s \,\mathsf{pass}\, T \implies i \,\mathsf{pass}\, T$. This is a classic testing characterisation result that opens the door to testing implementations for iocos-correctness. However, if we wanted to test $s \sqsubseteq_T i$ we would have to try all possible tests, since we do not know which are the tests that s passes.

An essential characteristic for a MBT framework is to be able to automatically produce a test suite from a model specification. For iocos we present this algorithm in Definition 10 which is a variation from the algorithm shown in [20]. The main difference in our algorithm is the inclusion of the $\oplus$ operator in tests. Let us note that if we have a deterministic specification[4]then $\oplus$ operator is applied to a singleton and therefore the resulting test does not really use such operator.

Definition 10. Let $(S, I, O, \rightarrow) \in LTS$ and $p \in S$. We denote with $\mathcal{T}(p)$ the set of valid tests from p by applying a finite number of recursive applications of one of the following non-deterministic choices:

1. $T = \checkmark \in \mathcal{T}(p)$.
2. If $a? \in \mathsf{ins}(p)$, then $T \in \mathcal{T}(p)$ where
$$T = a?; \bigoplus \{T_{p_{a?}} \mid p \xrightarrow{a?} p_{a?}\} + \sum_{\substack{x \in \mathsf{outs}(p) \\ x \neq \delta}} x; \bigoplus \{T_{p_x} \mid p \xrightarrow{x} p_x\} + \sum_{\substack{x \in O, x \neq \delta \\ x \notin \mathsf{outs}(p)}} x; \text{✗} \; + \; \delta; T_\delta(p)$$
3. If $\mathsf{ins}(p) = \varnothing$ then $T \in \mathcal{T}(p)$ where
$$T = \sum_{\substack{x \in \mathsf{outs}(p) \\ x \neq \delta}} x; \bigoplus \{T_{p_x} \mid p \xrightarrow{x} p_x\} + \sum_{\substack{x \in O, x \neq \delta \\ x \notin \mathsf{outs}(p)}} x; \text{✗} + \delta; T_\delta(p)$$

In all cases the tests T_p are chosen non-deterministically from the set $\mathcal{T}(p)$, $T_\delta(p) = \checkmark$ if $p \xrightarrow{\delta}$, and $T_\delta(p) = \text{✗}$ otherwise. □

The essential goal of the algorithm is to produce a test suite as tight as possible to the given specification. The rest of this section is devoted to prove the completeness of the algorithm for the iocos relation. There are two basic properties for the test suite: soundness (Proposition 3) and exhaustiveness (Proposition 4). The most basic property is soundness, meaning all tests from the set $\mathcal{T}(p)$ to be correct with respect p: p passes all tests in the set $\mathcal{T}(p)$.

Proposition 3. (***Soundness***)
Let $(S, I, O, \rightarrow) \in LTS$ and $p \in S$. Then $p \,\mathsf{pass}\, T$ for any $T \in \mathcal{T}(p)$.

[4] A behaviour is deterministic when for any $x \in L_\delta$, if $p \xrightarrow{x} p_1$ and $p \xrightarrow{x} p_2$ then $p_1 = p_2$.

Proof. by structural induction on the set of terms $\mathcal{T}(p)$. In base case, it trivially holds, since $p \,\textsf{pass}\, \checkmark$ for $T = \checkmark$. For recursive cases we have:

$T = a?; \bigoplus T_{p_{a?}} + \sum_{\substack{x \in \textsf{outs}(p) \\ x \neq \delta}} x; \bigoplus T_{p_x} + \sum_{\substack{x \in O, x \neq \delta \\ x \notin \textsf{outs}(p)}} x; \text{✗} + \delta; T_\delta(p)$**.** First let us note that $p \,\textsf{pass}\, \delta; T_\delta(p)$ trivially ($T_\delta(p) = \checkmark$ if $p \xrightarrow{\delta}$). By definition of the generator algorithm (Definition 10), $a? \in \textsf{ins}(p)$ and $T_{p_{a?}} \in \mathcal{T}(p_{a?})$ for any $p_{a?}$ such that $p \xrightarrow{a?} p_{a?}$. By induction hipothesis $p_{a?} \,\textsf{pass}\, T_{p_{a?}}$, therefore $p \,\textsf{pass}\, a?; \bigoplus T_{p_{a?}}$.
Similarly, if $x \in \textsf{outs}(P)$, $T_{p_x} \in \mathcal{T}(p_x)$ for any p_x such that $p \xrightarrow{x} p_x$. By induction hipothesis $p_x \,\textsf{pass}\, T_{p_x}$, therefore $p \,\textsf{pass} \sum_{x \in \textsf{outs}(p)} x; \bigoplus T_{p_x}$. If $x \notin \textsf{outs}(p)$ then $p \overset{x}{\not\longrightarrow}$, so $p \,\textsf{pass} \sum_{\substack{x \in O_\delta \\ x \notin \textsf{outs}(p)}} x; \text{✗}$
Finally, since p passes all addends from test T, $P \,\textsf{pass}\, T$.

$T = \sum_{\substack{x \in \textsf{outs}(p) \\ x \neq \delta}} x; \bigoplus T_{p_x} + \sum_{\substack{x \in O, x \neq \delta \\ x \notin \textsf{outs}(p)}} x; \text{✗} + \delta; T_\delta(p)$**.** This case is similar to the previous one. We only have to take into account that $\textsf{outs}(p) \neq \varnothing$. □

Proposition 4. (***Exhaustiveness***) Let $(S, I, O, \rightarrow) \in LTS$ and $p, q \in S$. If $\forall T \in \mathcal{T}(p) : q \,\textsf{pass}\, T$ then $p \sqsubseteq_T q$.

Proof. Let us prove the theorem by contradiction. Let us suppose $p \not\sqsubseteq_T q$, then there is a test $T \in \mathcal{T}_v$ such that $p \,\textsf{pass}\, T$ and $q \,\textsf{pa\not{s}s}\, T$. Let us prove that if there is a test T such that $p \,\textsf{pass}\, T$ and $q \,\textsf{pa\not{s}s}\, T$ there is a test $T_p \in \mathcal{T}(p)$ such that $q \,\textsf{pa\not{s}s}\, T_p$ by induction on T. The base case is when $T = \checkmark$ or $T = \text{✗}$; but in these cases there is nothing to prove since $q \,\textsf{pass}\, \checkmark$ and $p \,\textsf{pa\not{s}s}\, \text{✗}$.

So let us consider the recursive cases.

$T = T_1 \oplus T_2$**.** Then $p \,\textsf{pass}\, T_1$ or $p \,\textsf{pass}\, T_2$. In both cases we obtain the result by induction.

$T = \sum_{x \in O_\delta} x; T_x$**.** For any $x \in O_\delta$ let us consider a test T'_x as follows:

$x = \delta$**.** If $q \xrightarrow{\delta}$ then $T_x = \checkmark$, otherwise $T_x = \text{✗}$.

$x \in \textsf{outs}(q) \cap \textsf{outs}(p)$**.** If q_x $\textsf{pass}$ T_x for any q_x such that $q \xrightarrow{x} q_x$, let us consider $T_x = \checkmark$. Otherwise there is q_x such that $q \xrightarrow{x} q_x$ and $q_x \,\textsf{pa\not{s}s} T_x$. However, $p_x \,\textsf{pass}\, T_x$ for any p_x such that $p \xrightarrow{x} p_x$. So by induction, for any of those p_x there is a test $T_{p_x} \in \mathcal{T}(p_x)$ such that $q_x \,\textsf{pa\not{s}s}\, T_{p_x}$. Then let us consider $T'_x = \bigoplus\{T_{p_x} \mid p \xrightarrow{x} p_x\}$.

$x \in \textsf{outs}(q), x \notin \textsf{outs}(p)$**.** In this case $T'_x = \text{✗}$.

$x \notin \textsf{outs}(q), x \in \textsf{outs}(p)$**.** In this case $T'_x = \checkmark$.

So the test $T_p = \sum_{x \in O_\delta} x; T'_x$ satisfies that $T_p \in \mathcal{T}(p)$ and $q \,\textsf{pa\not{s}s}\, T_p$.

$T = a?; T_{a?} + \sum_{x \in O_\delta} x; T_x$**.** For $x \in O_\delta$ let us consider the test T'_x as in the previous case. Let us note that, since $p \,\textsf{pass}\, T$, $a? \in \textsf{ins}(p)$. Let us consider the test $T'_{a?}$ as follows:

$a? \in \textsf{ins}(q)$**.** $T'_{a?}$ is build in a similar way as in the previous case when $x \in \textsf{outs}(q) \cap \textsf{outs}(p)$.

$a? \notin \textsf{ins}(q)$**.** $T'_{a?} = \checkmark$.

So the test $T_p = a?; T'_{a?} + \sum_{x \in O_\delta} x; T'_x$ satisfies that $T_p \in \mathcal{T}(p)$ and $q \,\textsf{pa\not{s}s} T_p$ □

Theorem 3. (***Completeness***) Let $(S, I, O, \rightarrow) \in LTS$ and $p, q \in S$, $\forall T \in \mathcal{T}(p) : q \,\textsf{pass}\, T$ iff $p \,\textsf{iocos}\, q$. □

5 Conclusions and Future Work

MBT aims to offer a real and practical solution to effectively check the correctness of a system implementation against the model provided by the specification. At the core of any concrete MBT theory, framework or tool, it lies a conformance relation to decide if a given implementation is correct for the proposed specification.

For every concrete case study and industrial application, to select a suitable conformance relation is a decision that may depend on many ingredients: costs of implementation, security considerations, performance, context of application... We think it would be desirable to have a theory with the capacity to express conformance at different levels.

The research we present in this paper is a humble step in that direction. Instead of the classic approach based on linear semantics, we have used a conformance relation based on simulation semantics. The reason for this decision is that some recent research on process theory has shown [8,7] that the family of simulation semantics forms a backbone on the spectrum of semantics from which a hierarchy of layers of linear semantics can be derived in a systematic way. To further follow the applicability of this theoretical results, to the particular case of MBT with input-output transition systems, is one of lines of research we are currently working on.

Along the paper we have settled the basics results for a MBT theory based on a conformance simulation relation: The definition of iocos as a conformance relation has been motivated through examples; we have showed iocos to be an strict refinement of classic ioco relation for input-output labelled transition systems; a testing characterisation of iocos has been provided and also a test suite generation algorithm from specifications.

However, there are still well known issues in MBT that we need to address in our proposal. Regarding applicability we are specially interested in test selection and on-the-fly, or on-line, testing [22] which does not need to generate a priori test suites, but instead try to check dynamically the implementation under test.

As for test selection we are interested in the use metrics [6], but we think we can benefit from the new insights of recent works in the area (see for instance [17]). Moreover, the coinductive definition of iocos and the well known characterisations of simulations as games make our approach very suitable to further research the use of on-line testing with iocos.

References

1. Abramsky, S.: Observational equivalence as a testing equivalence. Theoretical Computer Science 53(3), 225–241 (1987)
2. de Alfaro, L.: Game models for open systems. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 269–289. Springer, Heidelberg (2004)
3. Alur, R., Henzinger, T.A., Kupferman, O., Vardi, M.Y.: Alternating refinement relations. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 163–178. Springer, Heidelberg (1998)

4. Belinfante, A.: Jtorx: A tool for on-line model-driven test derivation and execution. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 266–270. Springer, Heidelberg (2010)
5. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Software Eng. 4(3), 178–187 (1978)
6. Feijs, L.M.G., Goga, N., Mauw, S., Tretmans, J.: Test selection, trace distance and heuristics. In: Schieferdecker, I., König, H., Wolisz, A. (eds.) TestCom. IFIP Conference Proceedings, vol. 210, pp. 267–282. Kluwer (2002)
7. de Frutos-Escrig, D., Gregorio-Rodríguez, C., Palomino, M.: On the unification of process semantics: Equational semantics. Electronic Notes in Theoretical Computer Science 249, 243–267 (2009)
8. de Frutos-Escrig, D., Gregorio-Rodríguez, C.: (Bi)simulations up-to characterise process semantics. Information and Computation 207(2), 146–170 (2009)
9. van Glabbeek, R.J.: The Linear Time – Branching Time Spectrum I. In: Handbook of Process Algebra, pp. 3–99. Elsevier (2001)
10. van Glabbeek, R.J., Ploeger, B.: Correcting a space-efficient simulation algorithm. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 517–529. Springer, Heidelberg (2008)
11. Hennessy, M.: Algebraic Theory of Processes. MIT Press (1988)
12. Hierons, R.M., Bowen, J.P., Harman, M. (eds.): Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers. LNCS, vol. 4949. Springer, Heidelberg (2008)
13. Milner, R.: An algebraic definition of simulation between programs. In: Proceedings 2nd Joint Conference on Artificial Intelligence, pp. 481–489. BCS (1971)
14. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
15. Nicola, R.D., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science 34(1-2), 83–133 (1984), `http://www.sciencedirect.com/science/article/pii/0304397584901130`
16. Ranzato, F., Tapparo, F.: A new efficient simulation equivalence algorithm. In: LICS, pp. 171–180. IEEE Computer Society (2007)
17. Romero Hernández, D., de Frutos Escrig, D.: Defining distances for all process semantics. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 169–185. Springer, Heidelberg (2012)
18. Stokkink, G., Timmer, M., Stoelinga, M.: Talking quiescence: a rigorous theory that supports parallel composition, action hiding and determinisation. In: Petrenko, A.K., Schlingloff, H. (eds.) MBT. EPTCS, vol. 80, pp. 73–87 (2012)
19. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools 17(3), 103–120 (1996)
20. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, et al. (eds.) [12], pp. 1–38
21. Tretmans, J., Brinksma, E.: Torx: Automated model-based testing. In: Hartman, A., Dussa-Ziegler, K. (eds.) First European Conference on Model-Driven Software Engineering, pp. 31–43 (December 2003), `http://doc.utwente.nl/66990/`
22. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test., Verif. Reliab. 22(5), 297–312 (2012)
23. Veanes, M., Bjørner, N.: Alternating simulation and ioco. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 47–62. Springer, Heidelberg (2010)
24. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with spec explorer. In: Hierons et al, [12], pp. 39–76

Model Checking Distributed Systems against Temporal-Epistemic Specifications

Andreas Griesmayer[1] and Alessio Lomuscio[2]

[1] ARM, Cambridge, UK
[2] Imperial College London, London, UK

Abstract. Concurrency and message reordering are two main causes for the state-explosion in distributed systems with asynchronous communication. We study this domain by analysing ABS, an executable modelling language for object-based distributed systems and present a symbolic model checking methodology for verifying ABS programs against temporal-epistemic specifications. Specifically, we show how to map an ABS program into an ISPL program for verification with MCMAS, a model checker for multi-agent systems. We present a compiler implementing the formal map, exemplify the methodology on a mesh network use case and report experimental results.

1 Introduction

Significant advances have been made in the development of model checking techniques [5] for a variety of high level programming languages for distributed systems. These techniques support many features of distributed systems including *concurrency*, *modularity* and *object-orientation*. Much less attention has been given so far to the exploration of symbolic verification for *asynchronous message passing* using *futures* [1], a concept supported in modern programming languages like Java, Scala, or the latest C++ standard C++11. By using *futures*, message calls can be made without blocking the caller and messages may be delayed and reach their destinations out of order.

The aim of this paper is to put forward a verification technique targeting this specific class of systems. To make the investigation grounded in a concrete framework we base our analysis on ABS [10], an object-oriented modelling language for distributed systems. Advanced concurrency and synchronisation mechanisms, as well as a formally defined operational semantics and execution environment make ABS an ideal language for the formal modelling and analysis of distributed systems. A significant class of properties of distributed systems concern the *knowledge* that components have about the system they inhabit. To handle these specifications, we adopt techniques from verification of multi-agent systems (MAS). MAS are distributed systems in which the underlying components, or *agents*, interact with one another in order to maximise their own design objectives. MCMAS [15] is a model checker supporting a wide-range of specifications that commonly arise in MAS, including temporal-epistemic ones, and has been used successfully to verify a number of MAS scenarios, including agent-based web-services [16]. To utilise these techniques we provide a formal map from ABS to MAS, thus lifting the verification of temporal-epistemic MAS specifications to executable models in ABS.

We briefly summarise ABS, MCMAS, and the specification languages employed in the verification of MAS in Section 2. In Section 3 we define a formal map from a subset of ABS into ISPL and discuss how method invocation is handled. Experimental results and the implementation are described in Section 4 before we conclude in Section 5.

1.1 Related Work

While model checking has traditionally targeted the verification of finite systems, more recent work is concerned with constructs for software that lead to undecidable problems [19], including concurrency and recursion. These efforts have led to verification approaches for different programming paradigms like large subsets of C [4], Haskell [21], and Java [22]. Model checking is now routinely applied in well defined domains like driver development [23,2]. Furthermore, dedicated languages for modelling and verifying component based systems [3], protocols [9], etc., have been developed. Notwithstanding these important contributions, none of these languages provides direct support for asynchronous message passing, a fundamental feature in distributed systems. The semantics of ABS is formally defined and executed by means of the rewriting engine Maude [6], resulting in a well-defined semantics and enabling the user to access Maude's model checking capabilities. However, the interpretation of the semantic rules adds additional complexity and restricts model checking ABS to small sub-goals. By mapping ABS to MCMAS, we avoid some of this complexity and acquire the ability of verifying epistemic specifications.

In the context of distributed systems, traditional languages provide an unnatural transfer of control from the caller to the callee while waiting for the return from a method call. *Future*s [1] provide a fundamentally different control flow model that is centred on asynchronous message passing thereby enabling the calling process to carry on with its activities. This is a model that is much closer to Internet-based distributed systems. In this work we use ABS [10], a modelling language based on objects encapsulating their own processor, to analyse such systems. The models are executable, but because of their complexity, they can currently only be evaluated by using test case generation and simulation [12].

We show an application of the approach by using AODV, a well-understood protocol for ad-hoc message routing [18]. The AODV protocol has been analysed before through other model checkers [20,8] While these approaches can explore larger configurations than what we present here, their models are crafted manually and not generated automatically from executable programs. Closer to the approach we present here is CMC [17], which directly supports detailed, event-driven implementations in C and can be used to detect safety properties, including receipt of invalid packets. In contrast to these approaches, we here also present an automated translation from executable ABS programs into

$$\textit{(Asynch-Call)}\quad \frac{\overline{v} = [\![\overline{e}]\!]_{(a\circ l)}\ \ \mathit{fresh}(f)}{\begin{array}{c} ob(o, a, \{l \mid x = \hat{o}!m(\overline{e}); s\}, q) \\ \to ob(o, a, \{l \mid x = f; s\}, q) \\ \mathit{invoc}(\hat{o}, f, m, \overline{v})\ \mathit{fut}(f, \perp) \end{array}}
\qquad
\textit{(Bind-Mtd)}\quad \frac{p' = \mathit{bind}(o, f, m, \overline{v}, \mathit{class}(o))}{\begin{array}{c} ob(o, a, p, q)\ \mathit{invoc}(o, f, m, \overline{v}) \\ \to ob(o, a, p, q \cup p') \end{array}}$$

Fig. 1. Semantics rules for message passing in ABS

MCMAS and consider the asynchronous and concurrent nature of the underlying distributed systems. Based on this mapping, we perform symbolic model checking to verify temporal epistemic-properties, which requires the exploration of the full state space, hence inherently more expensive than safety checks.

2 Preliminaries

We give a short introduction to the concepts and techniques for the specification and verification framework of ABS. The semantic of the language is formally defined using Structural Operational Semantics (SOS) rules, which are given in rewrite logic and executable using the rewrite engine Maude [6]. We concentrate on the executable core language [10] with concurrent objects, in particular we give the details on the asynchronous message passing mechanisms. The behaviour of the system is analysed using temporal epistemic logic.

2.1 The Modelling Language ABS

ABS [10] uses objects that maintain local sets of processes to model independent communicating entities. Method calls are not accompanied with transfer of control to the called method. Instead, messages are transferred between caller and callee and a *future variable* serves as placeholder that can be polled at a later point in the execution to obtain the returned values. Messages *received* by the callee trigger the creation of new processes, which are added to its process set. ABS enforces strong data encapsulation, which means that variables can only be accessed locally or via method calls. Furthermore, each object performs local cooperative multitasking controlled by the *await*, *suspend* and *return* statements, where *return* publishes its return values in the *future* and terminates the execution, *suspend* unconditionally suspends the execution and makes an entry in a set of pending processes for a later resume, and *await* evaluates its condition and suspends the process while the result is false.

We follow the operational semantics of [10], given as SOS (Structural Operational Semantics) rules (Fig. 1). A state is represented as a *configuration (cn)*; a set of objects, messages and future variables. An *object* is a term $ob(o, a, p, q)$ where o is the object's identifier, a maps variable names (fields) to values, p is the currently executing process and q a set of suspended processes. A process consists of local variable bindings l and a list of statements s, denoted $\{l \mid s\}$. The full variable context is denoted as $a \circ l$, where variables in l hide variables in a if they have the same name. An *invocation message* $invoc(o, f, m, \overline{v})$ is sent to the *callee* o when the method m is called with parameters $\overline{v}$. The return message can be accessed via the *future* variable f. Variables can be combined with the usual Boolean and arithmetic operators to form expressions e where $[\![e]\!]_a$ denotes its evaluation with values from a. The details for the semantic rules that describe message passing are shown in Fig. 1 and described in the following:

(Asynch-Call) describes the execution of the statement $x = \hat{o}!m(\overline{e})$, which is on top of the statement list of the active process, followed by the statement list s. The premise of the SOS rule sets up the environment and states, where $\overline{v} = [\![\overline{e}]\!]_{(a \circ l)}$ is the evaluation of the arguments for the call in the current context and f is a fresh variable. In the consequence of the rule we find that the method call is removed from the statement list, and an invocation message $invoc(\hat{o}, f, m, v)$ and an empty future $fut(f, \bot)$ is created and assigned to the fresh variable f.

(Bind-Mtd) formalises the state change of the object receiving the *invoc* message by adding a new process p' to its idle set. $bind(o, f, m, \overline{v}, class(o))$ creates a new process to execute method m from object o with the parameters $\overline{v}$. The future variable f is assigned to a reserved local variable *destiny*, which is accessed later by the return statement.

Similar rules *(Return)* and *(Read-Fut)* define the return message of a call and access to the future variables.

2.2 Temporal-Epistemic Logic and Interpreted Systems

To reason about knowledge, we associate objects in ABS to *agents* in multi-agent systems (MAS). A popular semantics in MAS is that of interpreted systems [7], where agents interact with each other and their environment by means of actions. We adhere to standard naming conventions and characterise each agent $i \in \{1, \ldots, n\}$ in the system by finite sets of local states L_i and local actions Act_i. Actions are performed in compliance with a local protocol $P_i : L_i \to 2^{Act_i}$ specifying which actions may be performed in a given state. The environment in which agents live is modelled by a special agent E with a set of local states L_E, a set of local actions Act_E, and a local protocol P_E. A tuple $g = (l_1, \ldots, l_n, l_E) \in L_1 \times \ldots \times L_n \times L_E$, where $l_i \in L_i$ for each agent i and each $l_E \in L_E$, is a *global state* describing the system at a particular instant of time.

The evolution of the agents' local states is described by a function $t_i : L_i \times L_E \times Act_1 \times \ldots \times Act_n \times Act_E \to L_i$ which returns the next local state for agent i given the current local state of the agent, the current action and state L_E of the environment as well as all the agents' actions. Similarly, the evolution of the environment's local states is described by a function $t_E : L_E \times Act_1 \times \ldots \times Act_n \times Act_E \to L_E$, returning the next environment state given the current round of actions. It is assumed that in every state agents evolve simultaneously. The evolution of the global states of the whole system is described by a function $t : G \times Act \to G$, where $G \subseteq L_1 \times \ldots \times L_n \times L_E$ is the set of global states for the system reachable from a set of initial global states $I \subseteq G$, and $Act \subseteq Act_1 \times \ldots \times Act_n \times Act_E$ is the set of enabled joint actions. The function t is defined as $t(g, a) = g'$ if and only if for all $i, t_i(l_i(g), a) = l_i(g')$ and $t_E(l_E(g), a) = l_E(g')$, where $l_i(g)$ denotes the i-th component of global state g (corresponding to the local state of agent i). Finally, an interpreted system includes a set of atomic propositions AP together with a valuation function $V \subseteq AP \times G$. Formally, an *interpreted system* is defined as the tuple $IS = \langle (L_i, Act_i, P_i, t_i)_{i \in \{1,\ldots,n\}}, (L_E, Act_E, P_E, t_E), I, V \rangle$.

Temporal-Epistemic Logic. Interpreted systems provide a natural semantics to epistemic logic, or logic of knowledge, which is routinely used to specify MAS [7] like web-services [16]. We consider the following syntax defining our specification language:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid EG\varphi \mid E(\varphi U \varphi) \mid K_i\varphi, i \in \{1, \ldots, n\}$$

In the grammar above $p \in AP$ is an atomic proposition, and we have the usual negation and disjunction. Furthermore, $EX\varphi$ is read as *there is a global next state of computation in which φ holds*; $EG\varphi$ as *there exists a sequences of global states (or,* runs*) where φ holds in every state*, and $E(\varphi U \psi)$ as *there exists a run in which φ holds until ψ holds*; The knowledge operator $K_i\varphi$ is used to express that *agent i knows φ*. Based on this operators, we can define further useful operators to express often used properties. For instance, *there exists a run where φ is eventually true* ($EF\varphi$) can be written as $E(true\ U\varphi)$ and *φ holds on all runs in every state* ($AG\varphi$) is equivalent to *there is no run where φ does not hold* ($\neg EF\neg\varphi$).

Any interpreted system is associated to a model $M_{IS} = (W, R_t, \sim_1, \ldots, \sim_n, L)$ that can be used to interpret any formula φ. The set of possible worlds W is the set G of reachable global states. The temporal relation $R_t \subseteq W \times W$ relating two worlds (i.e., two global states) is defined by considering the transition function t of the corresponding IS: two worlds w and w' are such that $R_t(w, w')$ if and only if there exists a joint action $a \in Act$ such that $t(w, a) = w'$. The epistemic accessibility relations $\sim_i \subseteq W \times W$ are defined by considering the equality of the local components of the global states. Two worlds $w, w' \in W$ are such that $w \sim_i w'$ if and only if $l_i(w) = l_i(w')$ (i.e., two worlds w and w' are related via the epistemic relation $\sim_i$ when the local states of agent i in global states w and w' are the same). The labelling relation $L \subseteq AP \times W$ is defined in terms of the valuation relation V.

We write $(M, w) \models \varphi$ to represent that a formula φ is true at a world w in a Kripke model M. Temporal formulae are interpreted in M_{IS} as standard. For the epistemic operator K we have: $(M, w) \models K_i\varphi$ iff for all $w' \in W$ $w \sim_i w'$ implies $(M, w') \models \varphi$. We refer the reader to [7] for more details on this and related epistemic concepts widely discussed in the epistemic logic literature.

MCMAS [15] is a BDD-based model checker for the automatic verification of multi-agent systems. It provides ISPL (Interpreted Systems Programming Language) as an input language for modelling a MAS and expressing (amongst others) temporal and epistemic formulae as specifications of the system. The structure of an ISPL program allows the local states to be defined using *Boolean*, *bounded integer*, and *enumeration* variables. ISPL programs are closely related to interpreted systems; specifically, each ISPL program describes a unique interpreted system. MCMAS supports the verification for all formulae in the language above as well as others.

3 Mapping ABS Programs into ISPL

We define a map from a core subset of ABS to ISPL to verify the key innovative aspects of ABS. Given that any ISPL program uniquely defines an interpreted system, our map alternatively can be seen as the definition of a transition-based semantics for the subset of ABS we investigate.

To create a finite model for checking with MCMAS, we make a number of typical restrictions on the models to check. Specifically, we assume finite data types (Booleans, enumerations, bounded integers), introduce bounds on message and process queues (which also restricts recursion), and restrict object creation to an explicit *main* block. No *new* operator is used afterwards at run-time. Within these limitations, we support standard control flow and assignment statements, as well as the ABS-typical statements for local cooperative multitasking and message passing including data and object references. We refer to this bounded subset of the language as ABS_B. Extension by abstraction and introduction of parameterised techniques to handle infinite systems is left to future work.

The key correspondence we make is to associate objects in ABS_B to agents in ISPL and use the environment to handle message passing. Following this, we map ABS_B configurations to global states in interpreted systems and SOS rules to transitions, respectively. Our modelling of assignments and local conditionals follows the usual translation based on a current position in the control flow graph and corresponding updates to the variables. For more details see, e.g., the treatment of Boolean programs in [2]. We focus the discussion on message passing mechanisms and show that the semantic rules of Fig. 1 are preserved. The key element is the message buffer, which is encoded in the ISPL environment agent E. The buffer provides a bounded storage for n *invoc* messages, whose index also serves as value for future variables *fut*.

Definition 1 (Message Buffer). *A message buffer for n pending message calls in an ABS_B program is mapped to the ISPL environment E with local states $L_E \subseteq (B^1 \times \cdots \times B^n \times f \times agt \times tr)$. We write $b^i \in B^i$ for an entry in the buffer that holds a single call. It consists of a set of variables $b^i.\overline{v}$ and a status $b^i.stat$, which can be* `empty`*,* `o.m` *to represent a call to method m in object o,* `wr` *while waiting for a return, or* `pr` *while holding a return message. The remaining fields are used for communication where f holds the index of the active entry, agt the index of the connected agent, and tr the index of the next parameter to transfer.*

As previously remarked, objects in ABS_B are modelled as agents in ISPL, where each agent contains a bounded number of execution slots for processes.

Definition 2 (Object Agent). *Each object o in ABS_B is mapped to a corresponding agent in ISPL with local states $L_o \subseteq S^1 \times \cdots \times S^k \times stat \times tr$. We write $s^i \in S^i$ for a process slot with a pointer $s^j.exec$ to the next statement to execute, a reference $s^j.caller$ to the calling method buffer and local variables $s^j.\overline{v}$. The state of the object is stored in $stat$ and is either* `idle`*, initialising (*`init`*), or executing the processes in* $\texttt{slot}^{\text{k}}$. *The remaining variable tr holds the index of a slot that is communicating with the environment, or is 0 otherwise.*

We can give now the unique mapping between the ISPL model and the corresponding constructs from the ABS_B semantics. For a more succinct presentation, we use indices and exponents instead of the dot notation above when the meaning is clear from the context (e.g., we write $stat_E^i$ to refer to $l_E.b^i.stat$).

Definition 3 (Mapping Relation). *We define a mapping relation $\mu : cn \leftrightarrow L_E \times L_o$ between configurations and global states in the corresponding interpreted systems as*

$$\begin{aligned}
\mu(fresh(f)) &\leftrightarrow \quad stat_E^{val(f)} = \texttt{empty} \\
\mu(fut(f,\bot)) &\leftrightarrow \quad stat_E^{val(f)} \in \{\texttt{o\_m}, \texttt{wr}\} \\
\mu(fut(f,true)) &\leftrightarrow \quad stat_E^{val(f)} = \texttt{pr} \\
\mu(invoc(o,f,m,v)) &\leftrightarrow \quad stat_E^{val(f)} = \texttt{o\_m} \\
\mu(ob(o,a,p,q)) &\leftrightarrow \qquad stat_o = \texttt{slot}^{id(p)} \wedge exec_o^{id(p)} \neq 0 \wedge \forall_{j \in id(q)} exec_o^j \neq 0
\end{aligned}$$

where $val(f)$ is the value of the future variable f and $id(...)$ gives the (set of) indices of the process slots running respective processes. We denote the inverse as μ^{-1}.

Example 1. Fig. 2 gives an example of ABS code from a class *Node* of the AODV case study of Section 4 along with the relevant states of the control graph associated with the code. The special state ⓪ corresponds to an empty execution slot, states ① and ② represent that the execution of the object entered the corresponding method. The translation supports parameters, object references, global and local variables, maps and the usual assign and control flow statements.

3.1 Temporal Progress

Object agents and the environment progress simultaneously, synchronised by actions all participants need to agree on. We introduce the specific actions when needed and give the temporal progress of the system as transition relation for the object agent (t_o) and the environment (t_E) of the form

$$\begin{aligned}
t_o &= \{(l_o, l_E, a_E, a_o, l'_o) \mid \gamma\} \\
t_E &= \{(l_E, a_E, a_o, l'_E) \mid \gamma\}
\end{aligned}$$

```
Unit RREP(Node origin, Node src, Node dest, Int count){...}
Unit RREQ(Node origin, Node src, Node dest, Int count){
  if(src != this){
    rmap = store(rmap, origin, origin);
    dmap = store(dmap, origin, 1);
    route = lookup(rmap, src);
    if(route != null){
      distance = lookup(dmap, src);
      if (count < distance){
        rmap = store(rmap, src, origin);
        dmap = store(dmap, src, count);
    }}else{
        rmap = store(rmap, src, origin);
        dmap = store(dmap, src, count);
      }
    if(dest == this)
      origin!RREP(this, src, dest, 1);
    else{
      [...]
} } } } }
```

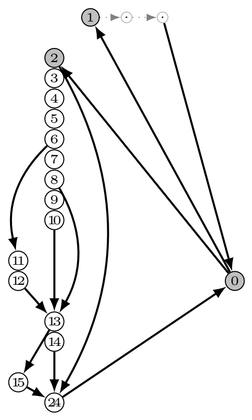


Fig. 2. AODV implementation in ABS along with the locations used in $exec_o^j$

where l_o and l_E are the current states of object agent and environment, a_E and a_o are the actions that are executed, and l'_o and l'_E are the new states for the object and the environment respectively. A transition only takes place if the environment and object agent agree on their actions. Formula γ is an update function that gives the values for the next state variables (primed) in terms of the variables of the current state (unprimed). In what follows, we give details on the update function γ representing message transfers involved in an ABS asynchronous method call.

Recall from Section 2 that protocols are used to determine which actions are enabled at a given state, whereas the local evolution function determines the successor state given some joint action. Consequently, the protocol of an object agent enables actions according to the *exec* pointer of the active execution slot, while the evolution function implements the variable updates for an executed action. For the message buffer, the protocol enables actions according to the status of its buffer entries, while the evolution function performs the actual data transfer and updates the variables.

To simplify the presentation, we give the temporal progress in the following as a single transition relation that contains the protocol function implicitly by defining that an action (and the corresponding transition) is only enabled if the unprimed variables match the current state. Because all communication is performed in one-to-one connections, we only describe the transitions concerned with this communication. However, the rules are instantiated for all agent combinations and agents not currently involved in a communication can perform local steps. Finally, we assume inertia for the variables, i.e, variables that are not explicitly changed by an assignment to their primed versions are held constant.

We give the transitions for sending a method call (SC) from the object to the environment. This is the first of four steps for a method call, which is followed by delivering the call (DC) from the environment to the callee, sending the return to the environment (SR) and receiving the return by the caller (RR). We will denote the caller object with o, the callee with $\hat{o}$, and the environment agent with the buffers as usual with E.

Sending a Method Call. The transfer of a method call $x := \hat{o}.m(\overline{v})$ from object agent o is performed in three phases: 1) initiating the transfer `(init)`, 2) transferring the data `(trf)`, and 3) closing the transfer `(cls)`.

The environment can initiate a transfer if there is an empty buffer ($stat_E^i = \texttt{empty}$) and no transfer is already performed ($tr_E = 0$). If this is the case, the action $\texttt{st\_o}_E$ is enabled. In the `(init)` step, the environment stores the object agent it communicates with in agt_E and the id of the buffer in f_E; tr_E is used as index of the parameter to transfer. In the second phase `(trf)`, action $\texttt{val\_x}_o$ transfers a value x from the caller, which is stored by action $\texttt{rc}_E$ in the buffer variable $v_E^{i,k} = x$, where k is the id of the transferred variable. Action $\texttt{ct}_E$ closes the transfer, where $\texttt{sto}_o$ indicates that the object agent stores a future variable for later reference ($rret_E^i = T$). If the object agent sends $\texttt{ign}_o$, no return value is required and the buffer can be freed after submitting the call to the callee ($rret_E^i = F$).

$$\begin{aligned}
t_E^{SC} = \{(l_E, \texttt{st\_o}_E, \hat{\texttt{o}}\texttt{\_m}_o, l'_E) \mid\ & stat_E^i = \texttt{empty} \wedge tr_E = 0 \wedge tr'_E = 1 && \texttt{(init)}\\
& \wedge {stat_E^i}' = \hat{\texttt{o}}\texttt{\_m} \wedge {agt_E}' = o \wedge f'_E = i\} \cup \\
\{(l_E, \texttt{rc}_E, \texttt{val\_x}_o, l'_E) \mid\ & agt_E = o \wedge tr_E = k \wedge f_E = i \wedge v_E^{i,k} = x && \texttt{(trf)}\\
& \wedge tr'_E = k + 1\} \cup \\
\{(l_E, \texttt{ct}_E, \texttt{sto}_o, l'_E) \mid\ & stat_E^i = \hat{\texttt{o}}\texttt{\_m} \wedge f_E = i \wedge f'_E = 0 && \texttt{(cls)}\\
& \wedge tr'_E = 0 \wedge {agt_E}' = 0 \wedge rret_E^i = T\} \cup \\
\{(l_E, \texttt{ct}_E, \texttt{ign}_o, l'_E) \mid\ & stat_E^i = \hat{\texttt{o}}\texttt{\_m} \wedge f_E = i \wedge f'_E = 0 && \texttt{(cls)}\\
& \wedge tr'_E = 0 \wedge {agt_E}' = 0 \wedge rret_E^i = F\}
\end{aligned}$$

The object agent o monitors the environment to signal the start of the communication with it by action $\texttt{st\_o}_E$, the caller at the same time indicates the object and method to call with $\hat{\texttt{o}}\texttt{\_m}_o$. The clause $stat_o = \texttt{slot}^\texttt{j}$ ensures that the process with the call is indeed currently executed and variable tr_o is set to j in the next state to mark an ongoing transfer (note that `(init)` is not enabled if tr_o is not 0 in the first place). In the second phase `(trf)`, action $\texttt{val\_x}_o$ transfers a value x from the caller, which is acknowledged by the environment by action $\texttt{rc}_E$. The index of the variable to transfer is given by environment variable tr_E. When all variables are transferred the transfer is closed in phase `(cls)` with the actions $\texttt{ct}_E$ and $\texttt{sto}_o$. In this step, object agent o stores f_E in a future variable of its local state and sets the execution pointer to the next statement.

$$\begin{aligned} t_o^{SC} = \{(l_o, l_E, \texttt{st\_o}_E, \texttt{ô\_m}_o, l'_o) \mid\ & exec_o^j = (x := \hat{o}.m(\overline{v})) \wedge tr_o = 0 && (\texttt{init}) \\ & \wedge stat_o = \texttt{slot}^\texttt{j} \wedge tr'_o = j\} \cup \\ \{(l_o, l_E, \texttt{rc}_E, \texttt{val\_x}_o, l'_o) \mid\ & exec_o^j = (x := \hat{o}.m(\overline{v})) \wedge tr_o = j \wedge v^k = x && (\texttt{trf}) \\ & \wedge stat_o = \texttt{slot}^\texttt{j} \wedge agt_E = o \wedge tr_E = k\} \cup \\ \{(l_o, l_E, \texttt{ct}_E, \texttt{sto}_o, l'_o) \mid\ & tr_o = j \wedge stat_o = \texttt{slot}^\texttt{j} && (\texttt{cls}) \\ & \wedge x_o^{j\prime} = f_E \wedge exec_o^{j\prime} = next() \wedge tr_o{}' = 0\} \end{aligned}$$

Method calls without assignment to a local variable (i.e. of the form $\hat{o}.m(\overline{v})$) do not store a reference to the future and therefore do not allow the caller to access any return value. In such a case, the buffer in the environment can be freed as soon as the callee is informed about the call. This is communicated to the environment by performing the action $\texttt{ign}_o$ in phase (`cls`) corresponds to second (`cls`) rule of the environment, which sets the `rret` field to false.

$$\begin{aligned} t_o^{SC} = \{(l_o, l_E, \texttt{st\_o}_E, \texttt{ô\_m}_o, l'_o) \mid\ & exec_o^j = (\hat{o}.m(v)) \wedge tr_o = 0 && (\texttt{init}) \\ & \wedge stat_o = \texttt{slot}^\texttt{j} \wedge tr'_o = j\} \cup \\ \{(l_o, l_E, \texttt{rc}_E, \texttt{val\_x}_o, l'_o) \mid\ & exec_o^j = (\hat{o}.m(v)) \wedge tr_o = j \wedge v^k = x && (\texttt{trf}) \\ & \wedge stat_o = \texttt{slot}^\texttt{j} \wedge agt_E = o \wedge tr_E = k\} \cup \\ \{(l_o, l_E, \texttt{ct}_E, \texttt{ign}_o, l'_o) \mid\ & tr_o = j \wedge stat_o = \texttt{slot}^\texttt{j} && (\texttt{cls}) \\ & \wedge exec_o^{j\prime} = next() \wedge tr_o{}' = 0\} \end{aligned}$$

Note that variable agt_E ensures the strict one to one connection for a transfer.

Similar protocols were defined for *delivering* the call to the callee, and sending the return message back to the caller. Correct synchronisation between agent and environment are essential for these operations. In contrast, ABS scheduling statements like `await`, which suspends the current process if an required return message has not arrived yet, does not require an action from the environment and is therefore implemented as local transition.

Partial Orders. The operational semantics of ABS does not impose any constraints on the execution order of statements in different objects or on the delivery order of sent messages. A full examination of the state space therefore involves the execution of all possible orders of the co-enabled statements, which leads to a state space and execution path explosion that is hard to handle even with succinct symbolic representations like BDDs. To reduce the data structures to a manageable size we employ *partial order reduction* (POR), by identifying partial orders of statements under which the same properties are fulfilled. For epistemic logic, however, the partial orders of actions that change the state must not be reduced even if the sequence of visible events is equivalent [14]. This is because the knowledge operators of epistemic logic consider the reachable state space to determine what an agent can know about the state of the other components. A reduction of the reachable states thus wrongly increases the computed knowledge of an agent.

To apply POR, we use the strong data encapsulation of ABS, which prevents information from being exchanged during method execution, and evaluate the epistemic properties over the full execution only at states where variables can be accessed, resp. when the agent is idle and can receive method calls to read a variable. This decision allows us to combine the statements of an execution block into a *macro step* [11] and remove the interleavings that stem from concurrent execution of methods from different agents. We furthermore treat the transitions for message passing given above as atomic blocks.

3.2 Preservation of ABS Semantics

The attention in the rest of this section is given to showing that the mapping preserves the semantics for the described bounded subset of the language ABS_B. An essential condition for that to happen is that the bounds on message buffers and execution slots are not overrun. To monitor this we introduce a flag *overrun* to the resulting ISPL file that is set to true when a buffer is requested but none is free. Checking for sufficient buffer sizes now corresponds to checking for $AG\neg overrun$, which holds if the bounds suffice.

Theorem 1 (Preservation of ABS semantics). *An ABS_B program P satisfies an epistemic specification ϕ iff $\mu(P) \models (\phi \wedge AG\neg overrun)$, where $\mu(P)$ is the interpreted system corresponding to the program P.*

Proof (sketch). Given sufficiently high bounds for the execution slots and buffers, executions in ISPL preserve the ABS semantics. This preservation can be shown by following the construction in [13], that is by establishing a Galois connection between the ABS operational semantics and ISPL transitions. We sketch the proof for the semantic rules from Fig. 1. First, we show that for a configuration c and an ISPL state u, $c \xrightarrow{\textit{(Asynch-Call)}} \mu^{-1}(u)$ if and only if $\mu(c) \xrightarrow{t_o \circ t_E} u$.

From the premise and left hand side of *(Asynch-Call)* we get $c = ob(o, a, \{l \mid x = \hat{o}!m(\overline{e}); s\}, q)$ with $\overline{v} = [\![\overline{e}]\!]_{(a \circ l)}$ and $fresh(f)$. Using $p := x = \hat{o}!m(\overline{e})$ and Def. 3 gives $\mu(c) = (stat_o = \mathtt{slot}^{id(p)} \wedge exec_o^{id(p)} = (x = \hat{o}.m(\overline{e})) \wedge next() = s)$, which corresponds to the states enabling t_o^{SC} and t_E^{SC}. Thus, we have that *(Asynch-Call)* is enabled if and only if t_o^{SC} and t_E^{SC} are enabled. The execution of all stages in t_o^{SC} and t_E^{SC} results in $u = (stat_E^{val(f)} = \hat{\mathtt{o}}\_\mathtt{m} \wedge v_E^{val(f)} = \overline{v} \wedge x_o^{val(f)} = val(f))$. The result of *(Asynch-Call)* and a subsequent store of the future as local variable gives $ob(o, a, \{x = f \mid s\}, q) \wedge invoc(\hat{o}, f, m, v) \wedge fut(f, \bot)$. Def. 3 shows that this result indeed is $\mu^{-1}(u)$, which concludes the proof.

The connection for *(Bind-Mtd)* and transitions $t_o^{DC} \circ t_E^{DC}$ is analogous where $p' = bind(o, f, m, \overline{v}, class(o)))$ is the instantiation of a new process and corresponds to $exec_o^{id(p')} = start(m) \wedge caller_o^{id(p')} = val(f) \wedge v_o^{id(p')} = \overline{v}$.

Limitations. Observe that the bounds on message buffer and execution slots may lead to delays in sending or receiving message calls. In particular, the *free* and *bind* predicates from the premises of the *(Asynch-Call)* and *(Bind-Mtd)* rules are not enabled when no buffer, or execution slot, respectively, is free. While this limits the examined behaviour if the bounds cannot be increased, the checked runs and any counterexample returned are still valid. This implies that checking properties for existential formulae is sound but not complete if the bounds are not sufficient.

4 Implementation and Experimental Results

We implemented the mapping from ABS to ISPL as a new back-end (ABSMC) to the ABS compiler framework, which already provides the parsing, type checking and generation of the abstract syntax tree. ABS programs have an explicit main block to set up the model. The ABSMC back-end executes this block and creates an agent with a unique name for each `new` statement, where arguments to the constructor are evaluated and passed as constants to the agent to reduce the state space.

The finite set of object names is used as enumerations to handle object references. Further variable types are Boolean, bounded integer, and enumerations. To allow generic variables in ISPL, the ABS variable types are mapped to integer of appropriate size. ABS allows annotations to statements and classes to provide additional information to the model. We use the annotations define the initial instances of an object and to provide the specifications we want to verify. The ISPL file generated by the ABSMC back-end is directly passed to the MCMAS model checker, which computes the set of reachable states and evaluates the formulae. MCMAS also can compute counterexamples and witnesses in a number of cases. The experiments were performed on an Intel i5 machine with 3.5GHz and 8GB RAM running Linux.

4.1 Verification of AODV Routing

To evaluate the applicability of the approach, we verified a distributed system implementing the *Ad-hoc On-Demand Distance Vector* (AODV) routing algorithm for mesh networks [18]. The purpose of the algorithm is to establish paths between two nodes from a set of arbitrarily distributed nodes, where not all nodes are within reach of each other. The algorithm is distributed in the sense that there is no central node deciding the routes of the packets. Instead, every node only knows the next hop to a given destination.

Initially no information about reachable nodes or routing is available to the nodes. The nodes communicate using RREQ (route request) or RREP (route reply) messages that carry the *origin* of the message, the *source* and the *destination* of the route to establish, and a *hop* count for the length of the route. The algorithm proceeds in two phases. First, an RREQ message is broadcast; this is relayed by its recipients until the destination is reached. The route is then established along the path of the RREQ messages using RREP messages.

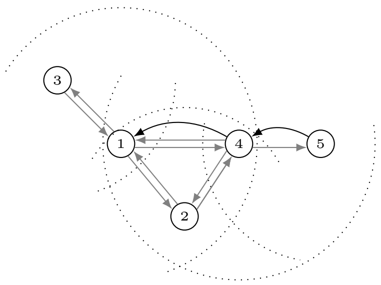


Fig. 3. AODV messages to establish a route between nodes 1 and 5

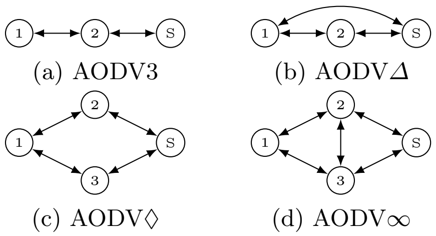


Fig. 4. Basic network topologies to verify properties of AODV

This is illustrated in Fig. 3, where RREQ broadcast messages are represented by straight grey lines, RREP by black curved lines, and dotted lines indicate the range of a node. Note that, since messages can be delayed, the first route that is established is not necessarily the shortest one. However, when several routes are available, the shortest is selected using the hop count.

We modelled the protocol in ABS using maps to store the next hop and the length of the route to a given node. Fig. 2 shows the ABS code executed at a node when receiving an RREQ message[1]. Upon receipt, if the source node is not the current node itself, the node first records that the origin of the message is a direct neighbour (distance 1), then it stores the route to the source via the origin unless there is already a shorter route known. If there is no route to the requested destination, the broadcast is relayed with increased hop count. Once a node with a route to the destination (or the actual destination itself) is reached, the second stage of the algorithm is performed by sending RREP messages towards the source of the route request. Otherwise, the message is broadcasted to its neighbours (omitted in the listing).

In our experiments we set up the different topologies from Fig. 4 and initiated the network by sending a request to node 1 (`N1`) to establish a route to node `S`. The compiler duly produced the resulting ISPL code that was fed to MCMAS as described earlier. We verified properties over the predicates `rXY` (representing that node `X` has a route to node `Y`), `rXZY` (node `X` has a route via node `Z` to node `Y`) and `finish` to denote that all messages have been delivered. We also added specifications to verify that, in all cases, routes are eventually established (represented as `AF r1S`) and that eventually the shortest route is found (`AF (finish` $\rightarrow$ `r1SS)` for Fig. 4b), even if intermediate routes might be longer (`EF r12S` in Fig. 4b).

Since MCMAS also supports an epistemic language we were further able to add specifications representing the information the agents have. In this context, we were able to check that when an agent sets a route, it knows that the next hop has a route to the destination (`AG (r12S -> K(N1, r2S))`). The topologies in Fig. 4c and Fig. 4d have different shortest routes. Which one is selected depends

[1] The examples are available at `http://www.doc.ic.ac.uk/~agriesma/ABSMC`

Table 1. Experimental results of the AODV case study

AODV3 | 250/24 | 20s | 50MB	
`AF r12S`	<1 T
`AG (r12S -> K(N1, r2S))`	<1 T

AODV$\varDelta$ | 250/24 | 32s | 54MB	
`AF r1SS`	<1 T
`AG (finish` $\rightarrow$ `r1SS)`	<1 T
`AF r12S`	10 F
`EF r12S`	2 T
`AG (r12S -> K(N1, r2S))`	<1 T
`AG (r1SS -> (! K(N1, r2S))`	3 F
`AG (lostRREP & r1SS -> (! K(N1, r2S))`	<1 T

AODV$\Diamond$ | 430/30 | 220s | 84MB	
`AF r12S`	64 F
`EF r12S`	19 T
`AF r1S`	<1 T
`AG (r12S -> K(N1, r2S))`	<1 T
`AG (r13S -> (! K(N1, r2S))`	36 F
`AG (lostRREP & r13S -> (! K(N1, r2S))`	<1 T

AODV∞ | 430/30 | 2262s | 351MB	
`AF r12S`	854 F
`EF r12S`	505 T
`AF r1S`	<1 T
`AG (r12S -> K(N1, r2S))`	<1 T
`AG (r13S -> (! K(N1, r2S))`	899 F

on the order in which the RREP messages arrive at the source. If one route is selected, the initial node does not know about the node on the other route; that is `AG (r12S -> K(N1, r3S))` is false, as MCMAS confirmed.

In the two topologies above, we might expect `AG (r12S -> !K(N1, r3S))` to hold. Our verification results showed, however, that this intuition is incorrect. The counterexample produced by MCMAS demonstrates that even in cases where a route via `N2` is selected, `N1` knows that `N3` has a valid route when it receives its RREP message. In 4d there are cases where `N1` can deduce some information even when this RREP is not taken into account and when its route goes via `N2`, namely the case when the route has 3 steps, in which case `N2` must route via `N3` and thus `N3` must have a route to `S`. Note that `N1` can deduce this knowledge solely from its local state and observations, and that they hold only for some of the possible orders of the messages. Model checking and examination of the counterexamples greatly helps to find such corner cases. To check for stability of the protocol, we added message loss to model 4a by allowing the environment to drop messages (in which case the predicate `lostRREP` is set to true). This corresponds to a change of topology where the corresponding connection is temporarily lost. The results show that for topologies with multiple routes, a route can be established even if one message is lost. We do, however, lose the certainty of finding the shortest route.

The results are summarised in Table 1, where for each topology the number of bits required for states/actions, the time for computing the reachable states, and the used memory are given. For each formula the result and computation time is given. We see that besides the number of nodes, which influences the number of possible states in the model, the number of connections is the main source of complexity. This is because the extra method calls from the additional messages increase the order of method calls that have to be considered, and thus the total number of reachable states. Note that the sizes of the checked systems are comparable to other work on AODV [20,8,17], while the used logic is more expressive and harder to check.

5 Conclusions

In this paper we presented a formal mapping between the modelling language for distributed systems ABS and interpreted systems, a formalism for multi-agent systems. We implemented the map into a compiler for the MCMAS model checker and evaluated the performance of the approach by studying a well-known networking protocol. While the approach requires us to provide bounds manually on the number of active processes, we were able to conclude the approach is mathematically sound and reasonably efficient. We are not aware of comparable literature on the subject as ABS models are currently only validated by simulation and test case generation [12]. Besides optimisations to the integration of the mapping with the model checker, our current direction of work in this line consists in providing automatic estimations on the required bounds for building the model as well as automatic abstraction techniques.

Acknowledgements. The authors would like to thank the members of the HATS (FP7-231620) project, and in particular Rudolf Schlatte and Jan Schäfer, for providing access and help to the ABS compiler framework.

The work reported in this paper was conducted at Imperial College London while Andreas Griesmayer was supported by the Marie Curie Fellowship “DiVerMAS” (FP7-PEOPLE-252184). Alessio Lomuscio acknowledges support from the UK Engineering and Physical Sciences Research Council through the Leadership Fellowship grant “Trusted Autonomous Systems” (EP/I00520X/1).

References

1. Baker Jr., H.C., Hewitt, C.: The incremental garbage collection of processes. In: Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages, pp. 55–59 (1977)
2. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)
3. Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.-H.: Compositional verification for component-based systems and application. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 64–79. Springer, Heidelberg (2008)
4. Clarke, E., Kroning, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
5. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (1999)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All about Maude - a high-performance logical framework: how to specify, program and verify systems in rewriting logic. LNCS, vol. 4350. Springer, Heidelberg (2007)

7. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about knowledge. The MIT Press, Cambridge (1995)
8. Fehnker, A., van Glabbeek, R., Höfner, P., McIver, A., Portmann, M., Tan, W.L.: Automated analysis of AODV using UPPAAL. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 173–187. Springer, Heidelberg (2012)
9. Holzmann, G.J.: The Spin Model Checker. Addison Wesley (2003)
10. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
11. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: A comparative analysis. In: PPPC 2009, pp. 11–20. ACM Press, New York (2009)
12. Leister, W., Bjørk, J., Schlatte, R., Griesmayer, A.: Verifying distributed algorithms with executable Creol models. In: PESARO 2011, pp. 1–6. IARIA (2011)
13. Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., Bensalem, S., Probst, D.: Property preserving abstractions for the verification of concurrent systems. Formal Methods in System Design 6(1), 11–44 (1995)
14. Lomuscio, A., Penczek, W., Qu, H.: Partial order reductions for model checking temporal epistemic logics over interleaved multi-agent systems. Fundamenta Informaticae 101(1), 71–90 (2010)
15. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: A model checker for the verification of multi-agent systems. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 682–688. Springer, Heidelberg (2009)
16. Lomuscio, A., Qu, H., Solanki, M.: Towards verifying contract regulated service composition. Journal of Autonomous Agents and Multi-Agent Systems 24(3), 345–373 (2010)
17. Musuvathi, M., Park, D.Y.W., Chou, A., Engler, D.R., Dill, D.L.: CMC: A pragmatic approach to model checking real code. SIGOPS Operating Systems Review, 36(SI), 75–88 (2002)
18. Perkins, C., Belding-Royer, E., Das, S.: Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental) (July 2003)
19. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Transactions on Programming Languages and Systems (TOPLAS) 22(2), 416–430 (2000)
20. De Renesse, F., Aghvami, A.H.: Formal verification of ad-hoc routing protocols using SPIN model checker. In: MELECON 2004, vol. 3, pp. 1177–1182. IEEE (2004)
21. Stolz, V., Huch, F.: Runtime verification of concurrent haskell programs. In: RV 2004. ENTCS, vol. 113, pp. 201–216. Elsevier Science Publishers (2005)
22. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering 10, 203–232 (2003)
23. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent linux device drivers. In: ASE 2007, pp. 501–504. ACM (2007)

Formal Verification of Distributed Branching Multiway Synchronization Protocols[⋆]

Hugues Evrard and Frédéric Lang

CONVECS Team, Inria Grenoble Rhône-Alpes and LIG
(*Laboratoire d'Informatique de Grenoble*), Montbonnot, France

Abstract. Distributed systems are hard to design, and formal methods help to find bugs early. Yet, there may still remain a semantic gap between a formal model and the actual distributed implementation, which is generally hand-written. Automated generation of distributed implementations requires an elaborate multiway synchronization protocol. In this paper, we explore how to verify correctness of such protocols. We generate formal models, written in the LNT language, of synchronization scenarios for three protocols and we use the CADP toolbox for automated formal verifications. We expose a bug leading to a deadlock in one protocol, and we discuss protocol extensions.

1 Introduction

Concurrent systems are hard to design, in particular distributed systems whose processes potentially run asynchronously, i.e., at independent speeds, possibly on remote machines. Formal methods, applied to formal system specifications, help to detect design flaws early in the development process. However, implementations are often hand-written, and a semantic gap may appear between a specification and its implementation. This can be palliated by tools which automatically generate a correct implementation from a formal specification.

We consider distributed systems consisting of several *tasks* that interact by *synchronization*. The specification of such systems will describe each task behavior as a nondeterministic process and the possible synchronizations between tasks through a parallel composition operator. As a particular specification language, we have in mind LOTOS NT (LNT for short) [5], a successor of LOTOS [16] and a variant of the E-LOTOS standard [17]. LNT has a general parallel composition operator [12] that enables *multiway synchronization* (also available in LOTOS), where a set of two or more tasks can synchronize altogether, and *m-among-n synchronization* (not available in LOTOS), where any subset of m tasks among a set of n tasks can synchronize altogether.

LNT is already equipped with formal verification tools packaged in the CADP toolbox [11]. In a close future, we would also like to automatically generate from

⋆ This work was partly funded by the French *Fonds national pour la Société Numérique* (FSN), Pôles Minalogic, Systematic and SCS (project OpenCloudware).

an LNT specification a distributed implementation consisting of one sequential process per task plus a synchronization protocol, as much distributed as possible to avoid the obvious bottleneck that a centralized synchronizer would represent in large distributed systems. Preserving the semantics of the specification is of major importance. We need elaborate protocols since classical synchronization barriers [8] cannot handle branching synchronizations, i.e., the situation where a task is ready to synchronize on several gates nondeterministically.

Several distributed synchronization protocols exist (see Section 2), many of them handling branching multiway synchronization, but not m-among-n synchronization. Some of these protocols have been proven correct either by demonstrating by hand the satisfaction of some properties, or by verifying by hand the behavior equivalence with an ideal synchronizer. To our knowledge, none of them has been verified using computer-assisted tools yet. We explore how protocols correctness can be verified using computer-assisted verification tools, which would provide better confidence in their correctness.

The contribution of this paper is the following. We selected three protocols that seemed most appropriate to handle LNT synchronization, respectively proposed by Sjödin [27], Parrow & Sjödin [23] and Sisto, Ciminiera & Valenzano [26] (respectively referred as Sjödin's, Parrow's and Sisto's protocol for short). For each of these three protocols, we generate formal specifications and use model checking to verify absence of deadlocks and livelocks, and equivalence checking to verify synchronization consistency and characterize precisely the semantic relation between the specification and the implementation. We claim that, under the hypotheses stated at the time of its publication, Parrow's protocol can lead to a deadlock, which we illustrate by an example and for which we propose a fix. At last, we discuss the limitations of the three protocols to handle m-among-n synchronization, and we propose some enhancements.

Paper Overview. Section 2 exposes the related work. Section 3 briefly presents the CADP toolbox. Section 4 introduces the three protocols under study. Section 5 explains how we generate formal specifications of protocols, and Section 6 lists the verifications we apply to these specifications. Section 7 discusses the results of protocol verifications, and describes the bug found in Parrow's protocol. Finally, Section 8 gives concluding remarks and directions for future work.

2 Related Work

There is an analogy between multiway synchronization and the Committee Coordination Problem [6] (CCP), where professors (tasks) may attend committees (synchronizations). A professor may attend any committee, a committee needs a predefined set of professors to be conveyed, and a professor can attend only one committee at a time. Committees sharing professors must be in mutual exclusion, and committees must be conveyed only if all professors are ready (readiness).

Chandy and Misra (C.&M.) propose a solution where the mutual exclusion is solved by mapping the problem to the Dining (or Drinking) Philosophers problem, and readiness is guaranteed by a shuffle of tokens [6]. Bagrodia presents the Event Manager (EM) algorithm, which uses a unique token cycling among committees to ensure mutual exclusion, and counters (of professors ready announcements and committee attendances) to guarantee readiness [1]. In the same paper, Bagrodia also proposes the Modified Event Managers (MEM) algorithm using the Dining Philosophers for mutual exclusion.

Bonakdarpour *et al.* address distributed implementations for the BIP framework [2]. Multiway synchronization is handled by a software layer, in which theoretically any protocol can be fitted. Their implementations use either a central synchronizer, a token-ring protocol (inspired by the EM algorithm of Bagrodia) or a mapping to the Dining Philosophers. They discuss the correctness of the derived implementation, but not of the protocols themselves.

LNT multiway synchronization differs slightly from the CCP in two ways. First, a single committee may be conveyed with different sets of professors: this is not a big deal, since we can declare new committees for every such set of professors and fall back to the CCP. Note however that we might face combinatorial explosion of committees, e.g. in the case of m-among-n synchronization. Second, a professor may be ready on a different subset of committees, depending on its current state. This extension to the CCP is addressed by C.&M. and Bagrodia: professors alert only committees they are ready on, but these still require mutual exclusion from all possible conflicting committees.

C.&M. and Bagrodia's protocols are based on solutions for synchronization in concurrency problems. At the same period, attempts to derive an implementation from a LOTOS specification lead to other solutions. Sisto *et al.* suggest a synchronization-tree based protocol [26]. In his thesis, Sjödin introduces a solution where committees directly lock professors [27], and a few years later Parrow & Sjödin propose a variation [23]. Although not in the framework of LOTOS, Perez *et al.* explore a very similar approach more recently [25].

In this paper, our main focus is protocol correctness. The solutions of C.&M., Bagrodia, and Perez are proven correct by satisfaction of properties. Sisto *et al.* discuss complexity but not correctness of their protocol. Sjödin demonstrates the equivalence between an ideal coordinator and his distributed solution; Parrow & Sjödin adopt the same approach but give only an overview of the proof. All these verifications are manual. To our knowledge, there was no attempt at verifying such protocols using automated verification tools.

We selected Sisto's, Sjödin's and Parrow's protocols in our study because, as they were designed to coordinate LOTOS synchronizations, they seemed most appropriate to handle also the case of LNT synchronizations efficiently. Regarding correctness, we verify not only the absence of livelocks and deadlocks, but we also compare the protocols' behavior with the expected reference behavior, which is obtained using reliable verification tools of CADP.

3 The CADP Toolbox

CADP (*Construction and Analysis of Distributed Processes*) [11] is a toolbox for modeling and verifying asynchronous systems. The CADP toolbox provides, among others, the following languages, models, and tools.

High-level languages allow concurrent systems to be modeled as processes running asynchronously and communicating by rendezvous synchronization on communication actions. Historically, LOTOS [16] was the main language of CADP. It combines algebraic abstract data types to model types and functions in an equational style, and a process algebra inheriting from CCS [21] and CSP [15] to model processes. In recent years, LNT [5] was developed, providing an easier syntax closer to mainstream imperative and functional programming languages. Models written in LNT can be verified using CADP, via an automated translation into LOTOS. The semantics of a LOTOS or an LNT program are defined as an LTS (*Labeled Transition System*) [21], that is a graph whose transitions between states are labeled by actions denoting value-passing communications.

Intermediate-level models are representations of systems between high-level languages and low-level models. As such, the EXP.OPEN 2.0 [18] language for networks of communicating LTSs consists of LTSs composed using various operators, including LOTOS and LNT parallel composition. EXP.OPEN 2.0 is a key component of CADP for compositional verification.

Low-level models are representations of LTSs. CADP provides the BCG (*Binary Coded Graph*) format to represent an LTS explicitly (as a set of states and transitions), and the OPEN/CÆSAR environment [9] to represent an LTS implicitly (as a set of types and functions, including functions for enumerating the successor transitions of a given state), for on-the-fly verification.

Temporal logics allow behavioral properties to be defined. The MCL language [20] combines the alternation-free μ-calculus together with regular formulas, primitives to handle data, and useful fairness operators of alternation 2.

Model checkers and *equivalence checkers* are also available in CADP. The EVALUATOR 4.0 model checker [20] allows an MCL formula to be checked on the fly on a system modeled in any language or format available in CADP, through the OPEN/CÆSAR interface. The BISIMULATOR 2.0 equivalence checker [19] allows the equivalence of two systems to be checked on the fly, modulo several equivalence relations, including strong [22], branching [29], safety [3] or weak trace [4] equivalences.

At last, CADP allows complex verification scenarios to be described succinctly using the intuitive language SVL (*Script Verification Language*) [10]. An SVL script is translated by the SVL compiler into a Bourne Shell script, which invokes the appropriate CADP tools automatically.

4 Overview of Synchronization Protocols

We consider a distributed system to be specified as several *tasks* which interact with each others by synchronous rendezvous on *gates*. A task is defined by an LTS of which transition labels are gate identifiers. A parallel composition expression defines for each gate which sets of tasks are synchronizable on that gate. In this paper, we also name a parallel composition of tasks a *synchronization scenario*.

Figure 1 illustrates a distributed system made of four tasks $t1$, $t2$, $t3$, and $t4$, which synchronize on gates A, B and C. Each task is represented by an LTS, the

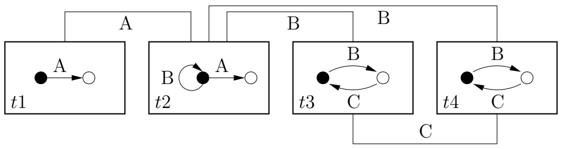


Fig. 1. A distributed system made of four tasks which synchronize on three gates

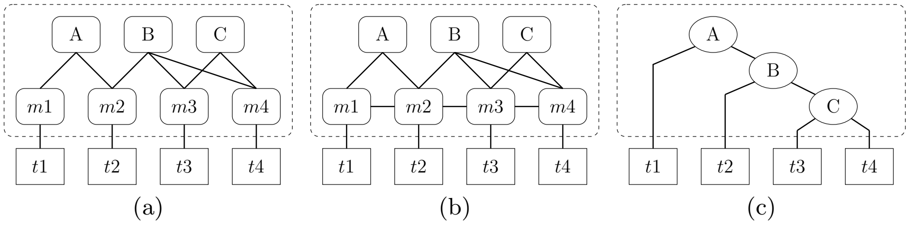


Fig. 2. Architecture of Sjödin's (a), Parrow's (b), and Sisto's (c) protocol

black point denoting the initial state. Possible synchronizations are represented by lines labeled with a gate identifier. For instance, a synchronization on B involves either $t2$ and $t3$, or $t2$ and $t4$.

A synchronization protocol must guarantee mutual exclusion of synchronizations which involve common tasks, and that a synchronization happens only when all involved tasks are actually ready on it. For instance in Figure 1, $t2$ may synchronize on B with either $t3$ or $t4$, but cannot synchronize with both at the same time. Once $t2$ has synchronized on A, it will never be ready to synchronize on B again, so no other synchronization on B may occur.

In the sequel we briefly describe how the three protocols under study fulfill these requirements. For a complete and detailed explanation of their internals, we refer to original publications of the protocols [27,23,26]. Note that LNT also enables data exchange during rendezvous on gates, and guards on data values. We leave those aspects for future work, focusing here on synchronization. In addition, we assume that the composition of tasks is static, i.e., we do not consider the dynamic creation and deletion of tasks.

Sjödin's Protocol Overview. A *mediator* process is associated to each task, and a *port* process is associated to each gate. Ready tasks send a message to their mediator, which lets know the relevant ports. When a port has received enough ready messages, it tries to lock all mediators involved in a synchronization. If it succeeds, then the synchronization occurs and the port sends a confirmation to all locked mediators, which announce to their task on which gate the synchronization occurred. Otherwise, if one of the mediators has already been locked and confirmed by another port, then negotiation is aborted and the port releases all mediators it has locked so far. To avoid deadlocks, all ports lock mediators in the same order [14]. Figure 2 (a) illustrates the architecture of Sjödin's protocol on the example of Figure 1.

Parrow's Protocol Overview. Parrow's protocol is based on Sjödin's and adopts almost the same architecture, see Figure 2 (b). The locking process is different: a port starts by locking the first mediator, which is then responsible for locking the next one, etc. When the last mediator is locked, it announces synchronization success to the port and to other involved mediators, which inform their tasks. However, if a mediator refuses the lock, it directly informs the port, and tells the list of locked mediators to release themselves. Compared to Sjödin's, Parrow's protocol mediators communicate with each other, and the locking process is less centralized.

Sisto's Protocol Overview. The protocol is very tied to LOTOS since it is structured as a composition tree obtained from a LOTOS expression. For instance, the parallel composition of Figure 1 can be expressed as the LOTOS expression "`t1 |[A]| (t2 |[B]| (t3 |[C]| t4))`". Figure 2 (c) illustrates the composition tree obtained, where leaves are tasks. Tasks announce which gates they are ready on to their upper *node*. Nodes may control one or several gates, in which case they collect ready announcements for these gates; for other gates they propagate ready messages to their father node. If both children of a node are ready on a gate controlled by the node, it starts to lock both subtrees down to the tasks. If a synchronization already occurred in a subtree, then the lock refusal is propagated upward. When the node which started the negotiation receives a refusal, it aborts the negotiation and unlocks the other subtree. If both subtrees accept the lock, then the node sends a confirmation message to both subtrees and the synchronization is achieved.

In the example of Figure 1, if $t2$, $t3$ and $t4$ are all ready on B, then the B node sends a lock to $t2$ and to the C node. Here the C node must choose if it propagates the lock of B to either $t3$ or $t4$, but must not synchronize both of them since $t3$ and $t4$ are interleaving on B. So each node is characterized by the gates it *controls* (for which it starts negotiations), and the gates it *synchronizes* (for which both children must be ready to propagate ready upward, and both children must be locked). These two sets may be different, and a node always synchronizes a gate it controls.

5 Formal Specification of Protocols

The three of the above protocols are made of *protocol processes* (namely *nodes* for Sisto's protocol, and *mediators* and *ports* for Sjödin's and Parrow's) which interact with tasks. In this section, we explain how, from a synchronization scenario, we automatically generate a formal specification of tasks, protocol processes, and their interaction. Figure 3 gives an overview of our specification and verification approach. The approach is generic, and may be used to verify other synchronization protocols.

We assume that the *high-level* specification of a synchronization scenario consists of an LTS stored in BCG format for each task, and of an EXP.OPEN expression for the parallel composition of tasks. Because Sisto's protocol is tied to a LOTOS expression, for the time being we assume the composition

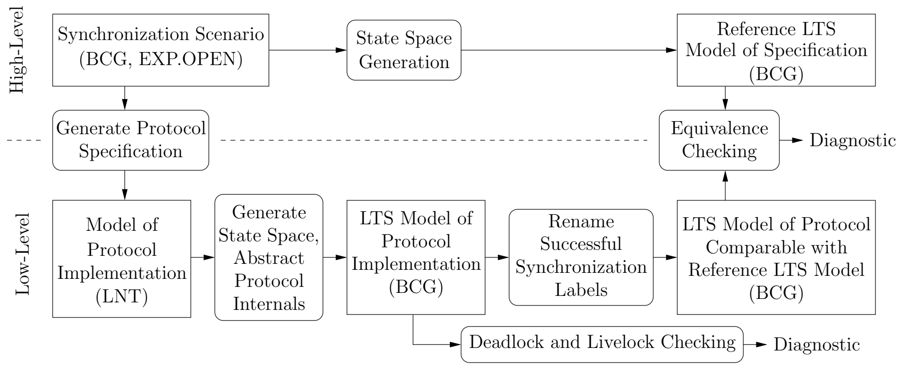

High-Level
Low-Level
Synchronization Scenario (BCG, EXP.OPEN)
State Space Generation
Reference LTS Model of Specification (BCG)
Generate Protocol Specification
Equivalence Checking
Diagnostic
Model of Protocol Implementation (LNT)
Generate State Space, Abstract Protocol Internals
LTS Model of Protocol Implementation (BCG)
Rename Successful Synchronization Labels
LTS Model of Protocol Comparable with Reference LTS Model (BCG)
Deadlock and Livelock Checking
Diagnostic

Fig. 3. Specification and verification steps in high and low level

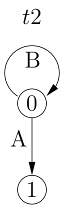
expression uses only LOTOS parallel composition[1]. This is our input to generate the *low-level* specification of the scenario, i.e., the model of the *implementation* of protocol processes, which manage synchronizations, and of tasks, which interact with protocol processes[2]. We write the low-level specification in LNT. We generate an LNT module for each task and for each protocol process. Moreover, a main module will compose tasks and protocol processes, along with LNT processes modeling the underlying network used in communications between tasks and protocol processes. Note that gates of high-level specification become data of message exchanges in low-level specification, and LNT gates in the low-level specification represent communication channels between low-level processes.

Low-Level Tasks. When a task is ready to synchronize on one or more gates, it must exchange messages with some protocol processes until it receives the confirmation of a successful synchronization. Therefore, a synchronization transition in the high-level specification becomes a sequence of messages exchanged between task and protocol processes, as defined by the protocol interface. For each protocol, and each task, a different low-level specification is generated depending on the protocol interface. For instance, Figure 4 illustrates the low-level specification of $t2$ for Parrow's protocol interface. The task first sends a synchronization request on gate `M`, along with the list of high-level gates it is ready on. If a synchronization succeeds, then the synchronized gate is stored in variable `sync_gate`, and the state to go next is selected accordingly.

Protocol Processes. Each protocol process has a generic behavior which is precisely described in the protocol's original publication. We just transcript this behavior in an LNT module, once for all. These modules take arguments to specialize their behavior according to the synchronization scenario. For instance,

[1] For instance, the EXP.OPEN composition expression corresponding to Figure 1 is: `"t1.bcg" |[A]| ("t2.bcg" |[B]| ("t3.bcg" |[C]| "t4.bcg"))`.

[2] Note that there is no relationship between the high and low levels of protocol models and the abstraction levels described in Section 3.



```
module task_t2 (data_types) is

process task_t2 [M: msg_channel] is    -- state 0
   var sync_gate: gate in
      M(request, {A, B});              -- send request to mediator
      M(confirm, ?sync_gate);          -- wait for confirmation
      case sync_gate in
         A -> task_t2_1[M]             -- synchro on A, go to state 1
      |  B -> task_t2[M]               -- synchro on B, go to state 0
      end case
   end var
end process

process task_t2_1 [M: msg_channel] is  -- state 1
   stop                                -- no outgoing transitions
end process

end module
```

Fig. 4. LNT code generated for a task using Parrow's protocol interface

in Sisto's protocol, arguments passed to nodes are the gates they control and the gates they synchronize. Moreover, in Sisto's protocol we introduce the `top_node` process which acts as a generic father for the root node.

Communications. The authors of the protocols assume that the underlying communication network is reliable (no messages are lost), and that tasks and protocol processes communicate via asynchronous message passing (i.e., sending and receiving the message are two distinct actions). Since LNT rendezvous is synchronous, we explicitly model communication buffers as LNT processes synchronizing with tasks and protocol processes.

Task and Protocol Process Composition. Finally, the main LNT process composes tasks, protocol processes and communication buffers in parallel. To model communication in a real network, this parallel composition uses only binary rendezvous between a communication buffer and either a task or a protocol process. Figure 5 illustrates the composition obtained from the example of Figure 1 for Sisto's protocol. For instance, a message from the `top_node` goes through a buffer via synchronization on `FOU` before reaching the destination node via a synchronization on `FOD`.

Tracing Successful Synchronization on Gate EXT. In order to track which high-level synchronizations are achieved using the protocol, we represent the "external world" with a low-level gate called EXT. Protocol processes report successful synchronization on a high-level gate by sending a message on EXT.

6 Verification of Protocols

Figure 3 and the SVL script of Figure 6 summarize our verification approach. From the main module of the low-level specification, we generate a raw low-level LTS. In this LTS, a transition is labeled by either a protocol message or a synchronization announcement on gate EXT (e.g, "`EXT !A`" for a synchronization

```
module main_sisto (data_types, top_node, node, buffer, task_t1, task_t2,
                   task_t3, task_t4) is

process main [EXT, F0U, F0D, F1U, F1D, F2U, F2D, F3U, F3D, F4U, F4D,
              F5U, F5D, F6U, F6D: message] is
   par
      F0U           -> top_node[EXT, F0U]
   || F0U, F0D      -> buffer[F0U, F0D]
   || F0D, F1U, F2U -> node[EXT, F0D, F1U, F2U]({A}, nil of gate_set)
   || F1U, F1D      -> buffer[F1U, F1D]
   || F2U, F2D      -> buffer[F2U, F2D]
   || F1D           -> task_t1[F1D]
   || F2D, F3U, F4U -> node[EXT, F2D, F3U, F4U]({B}, nil of gate_set)
   || F3U, F3D      -> buffer[F3U, F3D]
   || F4U, F4D      -> buffer[F4U, F4D]
   || F3D           -> task_t2[F3D]
   || F4D, F5U, F6U -> node[EXT, F4D, F5U, F6U]({C}, nil of gate_set)
   || F5U, F5D      -> buffer[F5U, F5D]
   || F6U, F6D      -> buffer[F6U, F6D]
   || F5D           -> task_t3[F5D]
   || F6D           -> task_t4[F6D]
   end par
end process -- main

end module
```

Fig. 5. Main LNT process of Sisto's low-level specification for the example of Figure 1

on gate A). For a given synchronization scenario and a given protocol, any possible order of protocol message exchanges and synchronization announcements is represented by a path in this LTS.

Our first transformation is hiding all internal protocol messages. In the low-level LTS obtained, all protocol messages are now labeled "`i`", which is the convention label for *internal actions* in LNT. We then perform the following verifications.

Livelock Detection. A livelock happens when low-level processes exchange messages indefinitely without agreeing on a synchronization, i.e., there exists somewhere in the low-level LTS a cycle of transitions which are only internal actions. Since this is the classical definition of a livelock, SVL comes with a built-in command to detect them (SVL actually calls the EVALUATOR4 tool of CADP with a predefined MCL formula that matches livelocks). If a livelock is detected, then a diagnostic, i.e., a path leading to a livelock, is stored in `diag_live.bcg`.

Deadlock Detection. Generally, a deadlock is defined by a state which has no outgoing transitions. Note that this can be an expected behavior: for instance in Figure 1 once $t1$ has synchronized on A, it reaches a deadlock. Such kind of situations trigger deadlocks in the low-level LTS too, and these deadlocks are not due to protocol errors.

Nonetheless, a protocol may get stuck into a deadlock while a synchronization *could* have been reached. This is unacceptable as the protocol must be able to offer a synchronization as long as one exists in the high-level model. In the low-level LTS, such a situation is characterized by a state from which there exists

```
(* Generate low-level LTS *)
"raw_lowlevel.bcg" = generation of "main.lnt";
(* Hide protocol messages *)
"lowlevel.bcg" = hide all but "EXT.*" in "raw_lowlevel.bcg";

(* Model checking: livelock and deadlock *)
"diag_live.bcg" = livelock of "lowlevel.bcg";
"diag_dead.bcg" = verify "deadlock.mcl" in "lowlevel.bcg";

(* Generate reference LTS from high-level spec *)
"reference.bcg" = generation of "composition.exp";
(* Rename synchronization announcements *)
"renamed.bcg" = total rename "EXT !\(.*\)" -> "\1" in "lowlevel.bcg";

(* Equivalence checking: branching, safety, weaktrace *)
"diag_branching.bcg" = branching comparison "renamed.bcg" == "reference.bcg";
"diag_safety.bcg" = safety comparison "renamed.bcg" == "reference.bcg";
"diag_weaktrace.bcg" = weak trace comparison "renamed.bcg" == "reference.bcg";
```

Fig. 6. Generic SVL script for verification operations

both: a sequence of internal actions which leads to a deadlock state; and another sequence which eventually contains a synchronization announcement. The MCL formula falsified by such protocol deadlocks is "`[true*] ((< "i"* > [true] false) implies [true* . not("i")] false)`". MCL is a rich language, and for the sake of brevity we do not explain the semantics of MCL constructions used in this formula. For more details, please refer to [20].

This MCL formula is stored in file `deadlock.mcl`, and it is evaluated on the low-level LTS (again, using EVALUATOR4 underneath). If a deadlock is found, the diagnostic is stored in `diag_dead.bcg`.

Synchronization Consistency. A synchronization protocol must not only be deadlock and livelock free, but it must also synchronize tasks correctly, so we finally have to verify synchronization consistency. We naturally use the *high-level LTS* generated from the high-level specification (the EXP.OPEN and tasks' BCG files) as a reference, i.e., we consider this LTS to actually represent which synchronizations are possible for this scenario, and we compare this high-level LTS with the low-level LTS using equivalence checking. To do so, labels of high-level and low-level LTS must be comparable. We rename low-level LTS labels using a simple regular expression, such that for instance the label "`EXT !A`" is renamed to "`A`". Now both LTSs have the same labels for task synchronizations, and the low-level one also contains internal actions representing protocol messages.

Several equivalence relations correspond to different ways of abstracting away internal actions. We use, in decreasing order of strength, the branching [29], safety [3] and weak trace [4] equivalence relations. The SVL script calls the BISIMULATOR tool, which compares LTSs and, in case of differences, provides a diagnostic showing a behavior possible in an LTS and not in the other.

7 Analysis of Protocol Verifications

The goal of model checking is finding bugs, rather than proving correctness. To this aim, we wrote a bench of 51 synchronization scenarios, trying to cover a

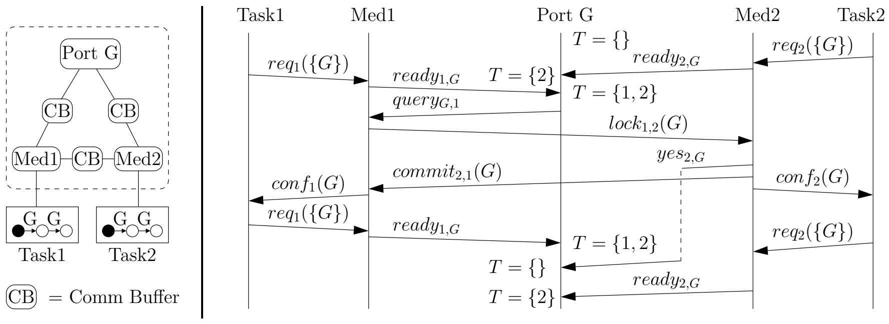


Fig. 7. Negotiation leading to a deadlock in Parrow's protocol

wide range of parallel compositions rather than a wide range of task behaviors. We thus focused on tasks containing up to no more than three states and ten transitions and we varied the number of concurrent tasks (from two up to four), the total number of gates (from zero up to four), the number of synchronized gates (from zero up to four), the number of gates simultaneously available in each task (from zero up to three), and the number of tasks synchronized on each gate (from one up to three). We obtain low-level LTSs with up to $500,000$ states and $1,200,000$ transitions.

Identification of Deadlocks in Parrow's Protocol. The test suite raised a design error that can lead to deadlocks. Figure 7 illustrates a scenario with two tasks Task1 and Task2 which synchronize two times on gate G. Figure 7 also exposes a negotiation leading to a deadlock after a first synchronization on G, whereas two synchronizations must occur. In the negotiation, we use the original notations of [23] for communication channels and data set names.

Each task notifies its own mediator with a request message to declare that it is ready on G. Mediators send ready messages to port G, which populates the set, called T, of tasks that are ready. When the port detects that both tasks are ready, it begins a negotiation by sending a *query* message to the mediator of Task1, called Med1. Med1 accepts the lock and sends a lock request to Med2. Med2 accepts the lock and sends a *yes* message to port G. We assume that this yes message is delayed (dashed line in Fig. 7), i.e., stored in a communication buffer and not consumed immediately.

Meanwhile, Med2 sends a *commit* message to Med1, and confirms successful synchronization on G to Task2. Med1 confirms to its task, and then both mediators receive new request messages. Med1 sends a ready message to port G, which accepts it. T is set to $T \cup \{1\}$, which actually leaves T unmodified.

Once the yes message from Med2 is received by port G, the set T is emptied so now $T = \{\}$. Med2 sends a ready notification, and T updates to $\{2\}$. In this situation, port G does not start a negotiation because T is not enough populated. However, ready requests of both mediators have already been received by the port. So we reach a deadlock, where a synchronization that could be successful (if the yes message had been received before the ready message of Med1) is not negotiated, and all protocol processes have stopped to communicate.

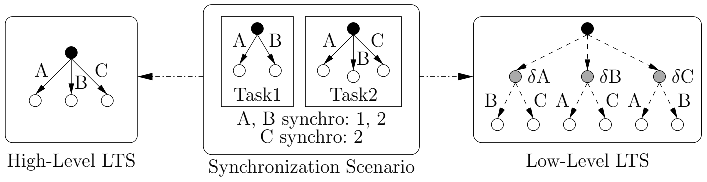


Fig. 8. High and low-level LTSs are not branching bisimilar

Parrow's protocol can be fixed by separating the set which stores ready announcements (let's call it N) and the set which is used for a negotiation (we keep T). Every time a ready message is received by a port, the corresponding task is inserted in N. Before starting a negotiation, involved tasks are moved from N to T. If a yes message is received, T is emptied (ready messages received before the yes were stored in N). If a no message is received, the task refusing the lock is removed from T, and remaining tasks of T are inserted back in N before T is emptied. Using our test suite, we verified that this modification corrects the design flaw without triggering new issues.

Equivalence of High and Low Level Models. Comparison between high-level and low-level models gives information about their relations in terms of execution trees and execution sequences, modulo a transitive closure of internal actions. Weak trace equivalence indicates that every execution sequence of the high-level model is also an execution sequence in the low-level model, and conversely. Stronger relations, such as safety equivalence and branching bisimulation, give information about execution trees, i.e., not only about sequences of executed actions, but also on the choices of alternative actions that can be offered in the intermediate states.

In the three of these protocols, we observe that the models are equivalent modulo safety equivalence[3] (which obviously implies weak trace equivalence), but are not branching bisimilar. This indicates that every execution tree of the high-level model is also an execution tree of the low-level model, and conversely, but that some execution subtrees of the low-level model may be strictly contained in the high-level model.

This is illustrated by Figure 8: for the sake of brevity we consider synchronizations involving only one task (here on gate C), which is a limit case of synchronization. In the high-level LTS, the choice between all three possible synchronizations is made from a single state. The low-level LTS contains interleavings of protocol messages, represented by dashed arrows. During negotiation, the next synchronization may require several messages to be progressively selected, i.e., we may reach states where a synchronization on a particular gate cannot occur anymore (the gate has been "discarded", marked δ on the figure), but the choice remains between other synchronizations. Such intermediate states, grayed on the figure, have no bisimilar state in the high-level LTS.

[3] In a manual proof, Sjödin and Parrow [24] use coupled simulation which, like safety equivalence, is a double simulation relation. We use the close but more standard safety equivalence, which is implemented in CADP.

For instance, consider Parrow's protocol on the scenario of Figure 8. If the first lock to happen is port A querying Med1, we reach a state where: if Med1 locks Med2 for A then A wins; or if port C locks Med2 then C wins (because both A and B need Med2, and C will not abort). Hence, we found a state where B will never happen but the choice between A or C remains.

Protocol Extension: *m*-among-*n* Synchronizations. So far, we considered high-level synchronizations to be specified by an EXP.OPEN expression using exclusively LOTOS parallel composition. In this section, we investigate how the protocols can manage *m*-among-*n* synchronizations offered by the LNT parallel composition operator. For instance, we write "`par A#2 in t1 || t2 || t3 end par`" to say that any group of 2 tasks among $t1$, $t2$ and $t3$ can synchronize on gate A. This cannot be directly expressed using LOTOS binary composition [12]. A way to implement this LNT operator is to flatten parallel composition by defining several sets of synchronizable tasks for each gate (*synchronization vectors*).

Sisto's protocol is so much tied to the tree structure of LOTOS expressions that it seems hard to make it manage *m*-among-*n* synchronizations without major design modifications.

Sjödin's protocol uses synchronization subtrees in its ready messages, and ports compose subtrees received from mediators to determine possible synchronizations. We replace these subtrees by simple lists of gates, and we give synchronization vectors as an argument to ports. Ports record ready announcements, and scan their synchronization vectors to detect possible synchronizations.

Parrow's protocol directly uses gate lists in its messages. However, each port is limited to only one synchronization vector, and in a locking sequence every mediator refers to this globally known vector to know what is the next mediator to lock. We extend port specifications to handle multiple synchronization vectors, and to send the relevant vector along lock requests. We also modify mediators to scan lock messages in order to know what is the next mediator to lock.

We verified that the modified versions of Sjödin's and Parrow's protocol still successfully handle synchronization scenarios of our test suite, plus a few more tests using *m*-among-*n* synchronizations.

Synchronization vectors are a direct and explicit way to express possible synchronizations. However, with nested parallel composition operators and *m*-among-*n* synchronizations, the number of synchronization vectors for a single gate can easily explode. To avoid this, we could use an equivalently expressive but more symbolic expression of possible synchronizations, such as the *synchronizers* proposed in the ATLANTIF intermediate model [28].

8 Conclusion and Future Work

In this paper, we presented how, for a given synchronization scenario and a given protocol, we can generate a formal LNT model of the implementation with asynchronous communication. Using the CADP verification toolbox, we spotted previously undetected deadlocks in Parrow's protocol (illustrated by an example), whereas we found no bug in Sjödin's and Sisto's protocols. To our knowledge, this is the first attempt at verifying such synchronization protocols using automated verification tools.

In their original formulation, the three protocols under study cannot handle the full LNT synchronization semantics. We believe Sisto's protocol cannot be easily extended because its behavior is closely related to the binary nature of the LOTOS parallel composition operator. On the other hand, we modified Sjödin's and Parrow's protocols such that possible synchronizations are now specified by synchronization vectors. These extended versions can handle the generality of LNT synchronization, and we verified that no new bugs were introduced.

The formal models of protocols will help us to decide which protocol to use for implementation. Nevertheless, before making our final decision, this work should be continued in several directions. First, we will study how data exchanges can be added to the protocols. Second, we could use the protocol models to precisely measure how many messages are required in each protocol to agree on a synchronization, depending on the number of tasks and the possible synchronizations between them. Moreover, we could use the performance evaluation features of CADP [7] to simulate communication latency between remote sites, and measure protocol performances directly on the formal models.

Finally, we will be able to develop a stand-alone compiler to generate a prototype distributed implementation of an LNT composition of tasks, as a family of remote task and protocol processes. The code for each task could be obtained by extending the EXEC/CÆSAR framework [13] of CADP (which currently generates sequential code simulating a concurrent or sequential process) to make it fit the synchronization protocol interface.

Acknowledgments. The authors warmly thank the Inria/CONVECS team members for useful discussions.

References

1. Bagrodia, R.: Process synchronization: Design and performance evaluation of distributed algorithms. IEEE Trans. on Software Engineering 15(9), 1053–1065 (1989)
2. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: From high-level component-based models to distributed implementations. In: Proc. of the 10th ACM International Conference on Embedded Software, pp. 209–218 (2010)
3. Bouajjani, A., Fernandez, J.C., Graf, S., Rodríguez, C., Sifakis, J.: Safety for Branching Time Semantics. In: Leach Albert, J., Monien, B., Rodríguez-Artalejo, M. (eds.) ICALP 1991. LNCS, vol. 510, pp. 76–92. Springer, Heidelberg (1991)
4. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A Theory of Communicating Sequential Processes. Journal of the ACM 31(3), 560–599 (1984)
5. Champelovier, D., Clerc, X., Garavel, H., Guerte, Y., McKinty, C., Powazny, V., Lang, F., Serwe, W., Smeding, G.: Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.8). Inria/CONVECS (2013)
6. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley (1988)
7. Coste, N., Garavel, H., Hermanns, H., Lang, F., Mateescu, R., Serwe, W.: Ten Years of Performance Evaluation for Concurrent Systems Using CADP. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part II. LNCS, vol. 6416, pp. 128–142. Springer, Heidelberg (2010)

8. Dijkstra, E.W.: The Structure of the "THE"-Multiprogramming System. Comm. of the ACM (1968)
9. Garavel, H.: OPEN/CAESAR: An Open Software Architecture for Verification, Simulation, and Testing. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 68–84. Springer, Heidelberg (1998)
10. Garavel, H., Lang, F.: SVL: a Scripting Language for Compositional Verification. In: Proc. of FORTE. Kluwer (2001)
11. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. STTT 15(2), 89–107 (2013)
12. Garavel, H., Sighireanu, M.: A Graphical Parallel Composition Operator for Process Algebras. In: Proc. of FORTE/PSTV. Kluwer (1999)
13. Garavel, H., Viho, C., Zendri, M.: System Design of a CC-NUMA Multiprocessor Architecture using Formal Specification, Model-Checking, Co-Simulation, and Test Generation. STTT 3(3), 314–331 (2001)
14. Havender, J.W.: Avoiding deadlock in multitasking systems. IBM Systems Journal 7(2), 74–84 (1968)
15. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
16. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization (1989)
17. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization (2001)
18. Lang, F.: EXP.OPEN 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-fly Verification Methods. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 70–88. Springer, Heidelberg (2005)
19. Mateescu, R., Oudot, E.: Bisimulator 2.0: An On-the-Fly Equivalence Checker based on Boolean Equation Systems. In: Proc. of MEMOCODE. IEEE (2008)
20. Mateescu, R., Thivolle, D.: A Model Checking Language for Concurrent Value-Passing Systems. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 148–164. Springer, Heidelberg (2008)
21. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
22. Park, D.: Concurrency and Automata on Infinite Sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104, pp. 167–183. Springer, Heidelberg (1981)
23. Parrow, J., Sjödin, P.: Designing a multiway synchronization protocol. Computer Communications 19(14), 1151–1160 (1996)
24. Parrow, J., Sjödin, P.: Multiway synchronization verified with coupled simulation. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 518–533. Springer, Heidelberg (1992)
25. Pérez, J.A., Corchuelo, R., Toro, M.: An order-based algorithm for implementing multiparty synchronization. Concurrency and Computation: Practice and Experience 16(12), 1173–1206 (2004)
26. Sisto, R., Ciminiera, L., Valenzano, A.: A protocol for multirendezvous of LOTOS processes. IEEE Trans. on Computers 40(4), 437–447 (1991)
27. Sjödin, P.: From LOTOS Specifications to Distributed Implementations. PhD thesis, Department of Computer Science, University of Uppsala, Sweden (1991)
28. Stöcker, J., Lang, F., Garavel, H.: Parallel Processes with Real-Time and Data: The ATLANTIF Intermediate Format. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 88–102. Springer, Heidelberg (2009)
29. van Glabbeek, R.J., Weijland, W.P.: Branching-Time and Abstraction in Bisimulation Semantics. In: Proc. of IFIP (1989)

An Abstract Framework for Deadlock Prevention in BIP*

Paul C. Attie[1], Saddek Bensalem[2], Marius Bozga[2], Mohamad Jaber[1], Joseph Sifakis[3], and Fadi A. Zaraket[4]

[1] Department of Computer Science, American University of Beirut, Beirut, Lebanon
[2] UJF-Grenoble 1 / CNRS VERIMAG UMR 5104, Grenoble, F-38041, France
[3] Rigorous System Design Laboratory, EPFL, Lausanne, Switzerland
[4] Department of Electrical and Computer Engineering, American University of Beirut, Beirut, Lebanon

Abstract. We present a sound but incomplete criterion for checking deadlock freedom of finite state systems expressed in BIP: a component-based framework for the construction of complex distributed systems. Since deciding deadlock-freedom for finite-state concurrent systems is PSPACE-complete, our criterion gives up completeness in return for tractability of evaluation. Our criterion can be evaluated by model-checking subsystems of the overall large system. The size of these subsystems depends only on the local topology of direct interaction between components, and *not* on the number of components in the overall system.

We present two experiments, in which our method compares favorably with existing approaches. For example, in verifying deadlock freedom of dining philosphers, our method shows linear increase in computation time with the number of philosophers, whereas other methods (even those that use abstraction) show super-linear increase, due to state-explosion.

1 Introduction

Deadlock freedom is a crucial property of concurrent and distributed systems. With increasing system complexity, the challenge of assuring deadlock freedom and other correctness properties becomes even greater. In contrast to the alternatives of (1) deadlock detection and recovery, and (2) deadlock avoidance, we advocate deadlock prevention: design the system so that deadlocks do not occur.

Deciding deadlock freedom of finite-state concurrent programs is PSPACE-complete in general [15, chapter 19]. To achieve tractability, we can either make our deadlock freedom check incomplete (sufficient but not necessary), or we can restrict the systems that we check to special cases. We choose the first option: a system meeting our condition is free of both local and global deadlocks, while a system which fails to meet our condition may or may not be deadlock free.

* The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement no. 288175 (CERTAINTY) and no 257414 (ASCENS).

We generalize previous works [2–4] by removing the requirement that interaction between processes be expressed pairwise, and also by applying to BIP [6], a framework from which efficient distributed code can be generated. In contrast, the model of concurrency in [2–4] requires shared memory read-modify-write operations with a large grain of atomicity. The full paper, including proofs for all theorems, is available on-line, as is our implementation of the method.

2 BIP – Behavior Interaction Priority

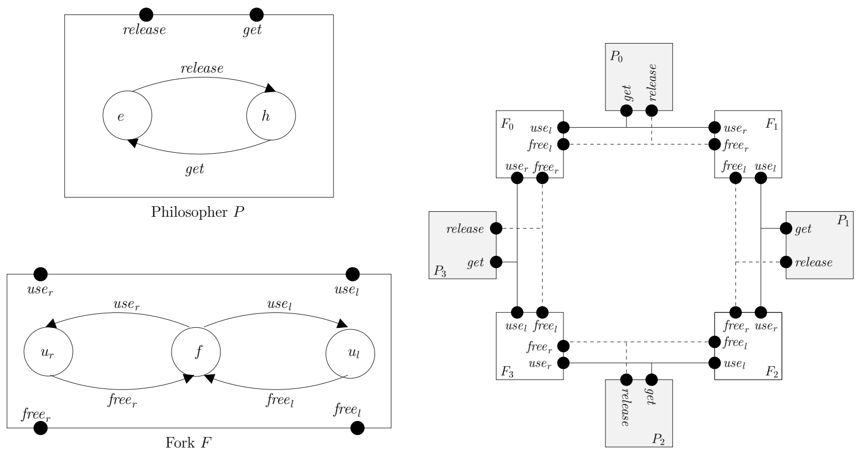
BIP is a component framework for constructing systems by superposing three layers of modeling: Behavior, Interaction, and Priority. A technical treatment of priority is beyond the scope of this paper. Adding priorities never introduces a deadlock, since priority enforces a choice between possible transitions from a state, and deadlock-freedom means that there is at least one transition from every (reachable) state. Hence if a BIP system without priorities is deadlock-free, then the same system with priorities added will also be deadlock-free.

Definition 1 (Atomic Component). *An* atomic component B_i *is a labeled transition system represented by a triple* $(Q_i, P_i, \rightarrow_i)$ *where* Q_i *is a set of* states*,* P_i *is a set of* communication ports*, and* $\rightarrow_i \subseteq Q_i \times P_i \times Q_i$ *is a set of* possible transitions*, each labeled by some port.*

For states $s_i, t_i \in Q_i$ and port $p_i \in P_i$, write $s_i \xrightarrow{p_i}_i t_i$, iff $(s_i, p_i, t_i) \in \rightarrow_i$. When p_i is irrelevant, write $s_i \rightarrow_i t_i$. Similarly, $s_i \xrightarrow{p_i}_i$ means that there exists $t_i \in Q_i$ such that $s_i \xrightarrow{p_i}_i t_i$. In this case, p_i is *enabled* in state s_i. Ports are used for communication between different components, as discussed below.

In practice, we describe the transition system using some syntax, e.g., involving variables. We abstract away from issues of syntactic description since we are only interested in enablement of ports and actions. We assume that enablement of a port depends only on the local state of a component. In particular, it cannot depend on the state of other components. This is a restriction on BIP, and we defer to subsequent work how to lift this restriction. So, we assume the existence of a predicate $enb^i_{p_i}$ that holds in state s_i of component B_i iff port p_i is enabled in s_i, i.e., $s_i(enb^i_{p_i}) = true$ iff $s_i \xrightarrow{p_i}_i$.

Figure 1(a) shows atomic components for a philospher P and a fork F in dining philosophers. A philosopher P that is hungry (in state h) can eat by executing *get* and moving to state e (eating). From e, P releases its forks by executing *release* and moving back to h. Adding the thinking state does not change the deadlock behaviour of the system, since the thinking to hungry transition is internal to P, and so we omit it. A fork F is taken by either: (1) the left philosopher (transition get_l) and so moves to state u_l (used by left philosopher), or (2) the right philosopher (transition get_r) and so moves to state u_r (used by right philosopher). From state u_r (resp. u_l), F is released by the right philosopher (resp. left philosopher) and so moves back to state f (free).



(a) Philosopher P and fork F atomic components.

(b) Dining philosophers composite component with four philosophers.

Fig. 1. Dining philosophers

Definition 2 (Interaction). *For a given system built from a set of n atomic components $\{B_i = (Q_i, P_i, \to_i)\}_{i=1}^n$, we require that their respective sets of ports are pairwise disjoint, i.e., for all i, j such that $i, j \in \{1..n\} \wedge i \neq j$, we have $P_i \cap P_j = \emptyset$. An* interaction *is a set of ports not containing two or more ports from the same component. That is, for an interaction a we have $a \subseteq P \wedge (\forall i \in \{1..n\} : |a \cap P_i| \leq 1)$, where $P = \bigcup_{i=1}^n P_i$ is the set of all ports in the system. When we write $a = \{p_i\}_{i \in I}$, we assume that $p_i \in P_i$ for all $i \in I$, where $I \subseteq \{1..n\}$.*

Execution of an interaction a involves all the components which have ports in a.

Definition 3 (Composite Component). *A* composite component *(or simply* component*) $B \triangleq \gamma(B_1, \dots, B_n)$ is defined by a composition operator parameterized by a set of interactions $\gamma \subseteq 2^P$. B has a transition system $(Q, \gamma, \to)$, where $Q = Q_1 \times \dots \times Q_n$ and $\to \subseteq Q \times \gamma \times Q$ is the least set of transitions satisfying the rule*

$$\frac{a = \{p_i\}_{i \in I} \in \gamma \qquad \forall i \in I : s_i \xrightarrow{p_i}_i t_i \qquad \forall i \notin I : s_i = t_i}{\langle s_1, \dots, s_n \rangle \xrightarrow{a} \langle t_1, \dots, t_n \rangle}$$

This inference rule says that a composite component $B = \gamma(B_1, \dots, B_n)$ can execute an interaction $a \in \gamma$, iff for each port $p_i \in a$, the corresponding atomic component B_i can execute a transition labeled with p_i; the states of components that do not participate in the interaction stay unchanged. Given an interaction $a = \{p_i\}_{i \in I}$, we denote by C_a the set of atomic components participating in a, formally: $C_a = \{B_i \mid p_i \in a\}$. Figure 1(b) shows a composite component consisting of four philosophers and the four forks between them. Each philosopher and its two neighboring forks share two interactions: $Get = \{get, use_l, use_r\}$ in which the philosopher obtains the forks, and $Rel = \{release, free_l, free_r\}$ in which the philosopher releases the forks.

Definition 4 (Interaction enablement). *An atomic component $B_i = (Q_i, P_i, \rightarrow_i)$ enables interaction a in state s_i iff $s_i \xrightarrow{p_i}_i$, where $p_i = P_i \cap a$ is the port of B_i involved in a. Let $B = \gamma(B_1, \ldots, B_n)$ be a composite component, and let $s = \langle s_1, \ldots, s_n \rangle$ be a state of B. Then B enables a in s iff every $B_i \in C_a$ enables a in s_i.*

The definition of interaction enablement is a consequence of Definition 3. Interaction a being enabled in state s means that executing a is one of the possible transitions that can be taken from s. Let enb^i_a denote the enablement condition for interaction a in component B_i. By definition, $enb^i_a = enb^i_{p_i}$ where $p_i = a \cap P_i$.

Definition 5 (BIP System). *Let $B = \gamma(B_1, \ldots, B_n)$ be a composite component with transition system $(Q, \gamma, \rightarrow)$, and let $Q_0 \subseteq Q$ be a set of initial states. Then (B, Q_0) is a BIP system.*

Figure 1(b) gives a BIP-system with philosophers initially in state h (hungry) and forks initially in state f (free).

Definition 6 (Execution). *Let (B, Q_0) be a BIP system with transition system $(Q, \gamma, \rightarrow)$. Let $\rho = s_0 a_1 s_1 \ldots s_{i-1} a_i s_i \ldots$ be an alternating sequence of states of B and interactions of B. Then ρ is an execution of (B, Q_0) iff (1) $s_0 \in Q_0$, and (2) $\forall i > 0 : s_{i-1} \xrightarrow{a_i} s_i$.*

A state or transition that occurs in some execution is called *reachable.*

Definition 7 (State Projection). *Let (B, Q_0) be a BIP system where $B = \gamma(B_1, \ldots, B_n)$ and let $s = \langle s_1, \ldots, s_n \rangle$ be a state of (B, Q_0). Let $\{B_{j_1}, \ldots, B_{j_k}\} \subseteq \{B_1, \ldots, B_n\}$. Then $s \upharpoonright \{B_{j_1}, \ldots, B_{j_k}\} \triangleq \langle s_{j_1}, \ldots, s_{j_k} \rangle$. For a single B_i, we write $s \upharpoonright B_i = s_i$. We extend state projection to sets of states element-wise.*

Definition 8 (Subcomponent). *Let $B \triangleq \gamma(B_1, \ldots, B_n)$ be a composite component, and let $\{B_{j_1}, \ldots, B_{j_k}\}$ be a subset of $\{B_1, \ldots, B_n\}$. Let $P' = P_{j_1} \cup \cdots \cup P_{j_k}$, i.e., the union of the ports of $\{B_{j_1}, \ldots, B_{j_k}\}$. Then the subcomponent B' of B based on $\{B_{j_1}, \ldots, B_{j_k}\}$ is as follows:*

1. $\gamma' \triangleq \{a \cap P' \mid a \in \gamma \wedge a \cap P' \neq \emptyset\}$
2. $B' \triangleq \gamma'(B_{j_1}, \ldots, B_{j_k})$

That is, γ' consists of those interactions in γ that have at least one participant in $\{B_{j_1}, \ldots, B_{j_k}\}$, and restricted to the participants in $\{B_{j_1}, \ldots, B_{j_k}\}$, i.e., participants not in $\{B_{j_1}, \ldots, B_{j_k}\}$ are removed.

We write $s \upharpoonright B'$ to indicate state projection onto B', and define $s \upharpoonright B' \triangleq s \upharpoonright \{B_{j_1}, \ldots, B_{j_k}\}$, where $B_{j_1}, \ldots, B_{j_k}$ are the atomic components in B'.

Definition 9 (Subsystem). *Let* (B, Q_0) *be a BIP system where* $B = \gamma(B_1, \dots, B_n)$*, and let* $\{B_{j_1}, \dots, B_{j_k}\}$ *be a subset of* $\{B_1, \dots, B_n\}$*. Then the subsystem* (B', Q'_0) *of* (B, Q_0) *based on* $\{B_{j_1}, \dots, B_{j_k}\}$ *is as follows:*

1. B' *is the subcomponent of* B *based on* $\{B_{j_1}, \dots, B_{j_k}\}$
2. $Q'_0 = Q_0 \restriction \{B_{j_1}, \dots, B_{j_k}\}$

Definition 10 (Execution Projection). *Let* (B, Q_0) *be a BIP system where* $B = \gamma(B_1, \dots, B_n)$*, and let* (B', Q'_0)*, with* $B' = \gamma'(B_{j_1}, \dots, B_{j_k})$ *be the subsystem of* (B, Q_0) *based on* $\{B_{j_1}, \dots, B_{j_k}\}$*. Let* $\rho = s_0 a_1 s_1 \dots s_{i-1} a_i s_i \dots$ *be an execution of* (B, Q_0)*. Then,* $\rho \restriction (B', Q'_0)$*, the projection of* ρ *onto* (B', Q'_0)*, is the sequence resulting from:*

1. *replacing each* s_i *by* $s_i \restriction \{B_{j_1}, \dots, B_{j_k}\}$*, i.e., replacing each state by its projection onto* $\{B_{j_1}, \dots, B_{j_k}\}$
2. *removing all* $a_i s_i$ *where* $a_i \notin \gamma'$

Proposition 1 (Execution Projection). *Let* (B, Q_0) *be a BIP system where* $B = \gamma(B_1, \dots, B_n)$*, and let* (B', Q'_0)*, with* $B' = \gamma'(B_{j_1}, \dots, B_{j_k})$ *be the subsystem of* (B, Q_0) *based on* $\{B_{j_1}, \dots, B_{j_k}\}$*. Let* $\rho = s_0 a_1 s_1 \dots s_{i-1} a_i s_i \dots$ *be an execution of* (B, Q_0)*. Then,* $\rho \restriction (B', Q'_0)$ *is an execution of* (B', Q'_0)*.*

Corollary 1. *Let* (B', Q'_0) *be a subsystem of* (B, Q_0)*. Let* s *be a reachable state of* (B, Q_0)*. Then* $s \restriction B'$ *is a reachable state of* (B', Q'_0)*. Let* $s \xrightarrow{a} t$ *be a reachable transition of* (B, Q_0)*, and let* a *be an interaction of* (B', Q'_0)*. Then* $s \restriction B' \xrightarrow{a} t \restriction B'$ *is a reachable transition of* (B', Q'_0)*.*

To avoid tedious repetition, we fix, for the rest of the paper, an arbitrary BIP-system (B, Q_0), with $B \triangleq \gamma(B_1, \dots, B_n)$, and transition system $(Q, \gamma, \rightarrow)$.

3 Characterizing Deadlock-Freedom

Definition 11 (Deadlock-freedom). *A BIP-system* (B, Q_0) *is* deadlock-free *iff in every reachable state* s *of* (B, Q_0)*, some interaction* a *is enabled.*

We assume in the sequel that each individual component B_i is deadlock-free, when considered in isolation, with respect to the set of initial states $Q_0 \restriction B_i$.

3.1 Wait-For Graphs

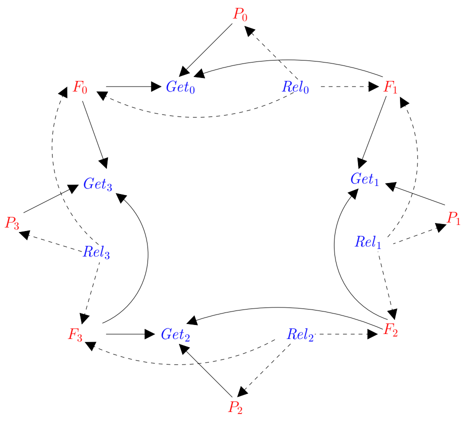
The wait-for-graph for a state s is a directed bipartite and-or graph which contains as nodes the atomic components $B_1, \dots, B_n$, and all the interactions γ. Edges in the wait-for-graph are from a B_i to all the interactions that B_i enables (in s), and from an interaction a to all the components that participate in a and which do not enable it (in s).

Definition 12 (Wait-for-graph $W_B(s)$). *Let $B = \gamma(B_1, \ldots, B_n)$ be a BIP composite component, and let $s = \langle s_1, \ldots, s_n \rangle$ be an arbitrary state of B. The* wait-for-graph *$W_B(s)$ of s is a directed bipartite and-or graph, where*

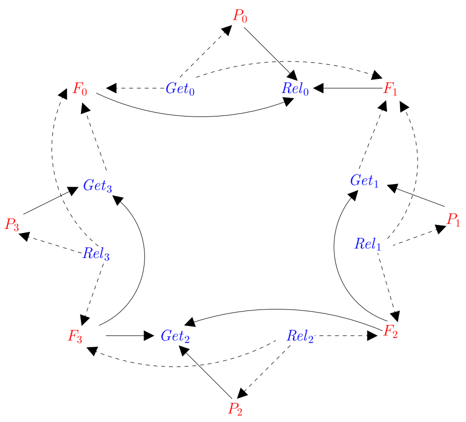
1. *the nodes of $W_B(s)$ are as follows:*
   (a) *the and-nodes are the atomic components B_i, $i \in \{1..n\}$,*
   (b) *the or-nodes are the interactions $a \in \gamma$,*
2. *there is an edge in $W_B(s)$ from B_i to every node a such that $B_i \in C_a$ and $s_i(enb^i_a) = true$, i.e., from B_i to every interaction which B_i enables in s_i,*
3. *there is an edge in $W_B(s)$ from a to every B_i such that $B_i \in C_a$ and $s_i(enb^i_a) = false$, i.e., from a to every component B_i which participates in a but does not enable it, in state s_i.*

A component B_i is an and-node since all of its successor actions (or-nodes) must be disabled for B_i to be incapable of executing. An interaction a is an or-node since it is disabled if any of its participant components do not enable it. An edge (path) in a wait-for-graph is called a wait-for-edge (wait-for-path). Write $a \to B_i$ ($B_i \to a$ respectively) for a wait-for-edge from a to B_i (B_i to a respectively). We abuse notation by writing $e \in W_B(s)$ to indicate that e (either $a \to B_i$ or $B_i \to a$) is an edge in $W_B(s)$. Also $B \to a \to B' \in W_B(s)$ for $B \to a \in W_B(s) \wedge a \to B' \in W_B(s)$, i.e., for a wait-for-path of length 2, and similarly for longer wait-for-paths.

Consider the dining philosophers system given in Figure 1. Figure 2(a) shows its wait-for-graph in its sole initial state. Figure 2(b) shows the wait-for-graph after execution of get_0. Edges from components to interactions are shown solid, and edges from interactions to components are shown dashed.



(a) Wait-for-graph in initial state.



(b) Wait-for-graph after execution of get_0.

Fig. 2. Example wait-for-graphs for dining philosophers system of Figure 1

3.2 Supercycles and Deadlock-Freedom

We characterize a deadlock as the existence in the wait-for-graph of a graph-theoretic construct that we call a *supercycle*:

Definition 13 (Supercycle). *Let $B = \gamma(B_1, \ldots, B_n)$ be a composite component and s be a state of B. A subgraph SC of $W_B(s)$ is a supercycle in $W_B(s)$ if and only if all of the following hold:*

1. *SC is nonempty, i.e., contains at least one node,*
2. *if B_i is a node in SC, then for all interactions a such that there is an edge in $W_B(s)$ from B_i to a:*
   (a) *a is a node in SC, and*
   (b) *there is an edge in SC from B_i to a,*
   that is, $B_i \to a \in W_B(s)$ implies $B_i \to a \in SC$,
3. *if a is a node in SC, then there exists a B_j such that:*
   (a) *B_j is a node in SC, and*
   (b) *there is an edge from a to B_j in $W_B(s)$, and*
   (c) *there is an edge from a to B_j in SC,*
   that is, $a \in SC$ implies $\exists B_j : a \to B_j \in W_B(s) \wedge a \to B_j \in SC$,

where $a \in SC$ means that a is a node in SC, etc. $W_B(s)$ is *supercycle-free* iff there does not exist a supercycle SC in $W_B(s)$. In this case, say that state s is supercycle-free.

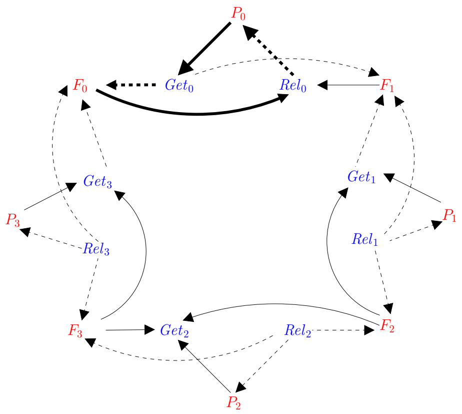


Fig. 3. Example supercycle for dining philosophers system of Figure 1

Figure 3 shows an example supercycle (with boldened edges) for the dining philosophers system of Figure 1. P_0 waits for (enables) a single interaction, Get_0. Get_0 waits for (is disabled by) fork F_0, which waits for interaction Rel_0. Rel_0 in turn waits for P_0. However, this supercycle occurs in a state where P_0 is in h and F_0 is in u_l. This state is not reachable from the initial state.

The existence of a supercycle is sufficient and necessary for the occurrence of a deadlock, and so checking for supercycles gives a sound and complete check for deadlocks. Write $SC \subseteq W_B(s)$ when SC is a subgraph of $W_B(s)$. Proposition 2 states that the existence of a supercycle implies a local deadlock: all components in the supercycle are blocked forever.

Proposition 2. *Let s be a state of B. If $SC \subseteq W_B(s)$ is a supercycle, then all components B_i in SC cannot execute a transition in any state reachable from s, including s itself.*

Proof sketch. Every interaction a that B_i enables is not enabled by some participant. By Defintion 4, a cannot be executed. Hence B_i cannot execute any transition.

Proposition 3 states that the existence of a supercycle is necessary for a local deadlock to occur: if a set of components, *considered in isolation*, are blocked, then there exists a supercycle consisting of exactly those components, together with the interactions that each component enables.

Proposition 3. *Let B' be a subcomponent of B, and let s be an arbitrary state of B such that B', when considered in isolation, has no enabled interaction in state $s{\restriction}B'$. Then, $W_B(s)$ contains a supercycle.*

Proof sketch. Every atomic component B_i in B' is individually deadlock free, by assumption, and so there is at least one interaction a_i which B_i enables. Now a_i is not enabled in B', by the antecedent of the proposition. Hence a_i has some outgoing wait-for-edge in $W_B(s)$. The subgraph of $W_B(s)$ induced by all the B_i and all their (locally) enabled interactions is therefore a supercycle.

We consider subcomponent B' in isolation to avoid other phenomena that prevent interactions from executing, e.g., conspiracies [5]. Now the converse of Proposition 3 is that absence of supercycles in $W_B(s)$ means there is no locally deadlocked subsystem. Taking $B' = B$, this implies that B is not deadlocked, and so there is at least one interaction of B which is enabled in state s.

Corollary 2. *If, for every reachable state s of (B, Q_0), $W_B(s)$ is supercycle-free, then (B, Q_0) is deadlock-free.*

Proof sketch. Immediate from Proposition 3 (with $B' = B$) and Definition 11.

3.3 Structural Properties of Supercycles

We present some structural properties of supercycles, which are central to our deadlock-freedom condition.

Definition 14 (Path, path length). *Let G be a directed graph and v a vertex in G. A path π in G is a* finite *sequence $v_1, v_2, \ldots, v_n$ such that (v_i, v_{i+1}) is an edge in G for all $i \in \{1, \ldots, n-1\}$. Write $path_G(\pi)$ iff π is a path in G. Define $first(\pi) = v_1$ and $last(\pi) = v_n$. Let $|\pi|$ denote the length of π, which we define as follows:*

– *if π is simple, i.e., all v_i, $1 \le i \le n$, are distinct, then $|\pi| = n - 1$, i.e., the number of edges in π*
– *if π contains a cycle, i.e., there exist v_i, v_j such that $i \neq j$ and $v_i = v_j$, then $|\pi| = \omega$ (ω for "infinity").*

Definition 15 (In-depth, Out-depth). *Let G be a directed graph and v a vertex in G. Define the in-depth of v in G, notated as $in\_depth_G(v)$, as follows:*

– *if there exists a path π in G that contains a cycle and ends in v, i.e., $|\pi| = \omega \wedge last(\pi) = v$, then $in\_depth_G(v) = \omega$,*
– *otherwise, let π be a longest path ending in v. Then $in\_depth_G(v) = |\pi|$.*

Formally, $in\_depth_G(v) = (\mathrm{MAX}\ \pi : path_G(\pi) \wedge last(\pi) = v : |\pi|)$.

Likewise define $out\_depth_G(v) = (\mathrm{MAX}\ \pi : path_G(\pi) \wedge first(\pi) = v : |\pi|)$, the out-depth of v in G, i.e., we consider paths starting (rather than ending) in v.

We use $in\_depth_B(v, s)$ for $in\_depth_{W_B(s)}(v)$, and also $out\_depth_B(v, s)$ for $out\_depth_{W_B(s)}(v)$.

Proposition 4. *A supercycle SC contains no nodes with finite out-depth.*

Proof sketch. By contradiction. Let v be a node in SC with finite out-depth. Hence all outgoing paths from v end in a sink node. By assumption, all atomic components are individually deadlock-free, i.e., they always enable at least one interaction. Hence these sink nodes are all interactions, and therefore they violate clause 3 in Definition 13.

Proposition 5. *Every supercycle SC contains at least one cycle.*

Proof sketch. Suppose not. Then SC is an acyclic supercycle. Hence every node in SC has finite out-depth, which contradicts Proposition 4.

Proposition 6. *Let $B = \gamma(B_1, \dots, B_n)$ be a composite component and s a state of B. Let SC be a supercycle in $W_B(s)$, and let SC' be the graph obtained from SC by removing all vertices of finite in-depth and their incident edges. Then SC' is also a supercycle in $W_B(s)$.*

Proof sketch. By Proposition 5, SC' is nonempty. Thus SC' satisfies clause (1) of Definition 13. Let v be an arbitrary vertex of SC'. Hence v has infinite in-depth, and therefore so do all of v's sucessors in SC. Hence all of these successors are in SC'. Hence every vertex v in SC' has successors in SC' that satisfy clauses (2) and (3) of Definition 13.

4 A Global Condition for Deadlock Freedom

Consider a reachable transition $s \xrightarrow{a} t$ of (B, Q_0). Suppose that the execution of this transition creates a supercycle SC, i.e., $SC \not\subseteq W_B(s) \wedge SC \subseteq W_B(t)$. The only components that can change state along this transition are the participants of a, i.e., the $B_i \in C_a$, and so they are the only components that can cause a supercycle to be created in going from s to t. There are three relevant possibilities for each $B_i \in C_a$:

1. B_i has finite in-depth in $W_B(t)$: then, if $B_i \in SC$, it can be removed and still leave a supercycle SC', by Proposition 6. Hence SC' exists in $W_B(s)$, and so B_i is not essential to the creation of a supercycle.
2. B_i has finite out-depth in $W_B(t)$: by Proposition 4, B_i cannot be part of a supercycle, and so $SC \subseteq W_B(s)$.
3. B_i has infinite in-depth and infinite out-depth in $W_B(t)$: in this case, B_i is possibly an essential part of SC, i.e., SC was created in going from s to t.

We thus impose a condition which guarantees that only case 1 or case 2 occur.

Definition 16 ($\mathcal{DFC}(a)$). *Let $s \xrightarrow{a} t$ be a reachable transition of BIP-system (B, Q_0). Then, in t, the following holds. For every component B_i of C_a: either B_i has finite in-depth, or finite out-depth, in $W_B(t)$. Formally,*

$$\forall B_i \in C_a : \mathit{in\_depth}_B(B_i, t) < \omega \vee \mathit{out\_depth}_B(B_i, t) < \omega.$$

To proceed, we show that wait-for-edges not involving some interaction a and its participants $B_i \in C_a$ are unaffected by the execution of a. Say that edge e in a wait-for-graph is *B_i-incident* iff B_i is one of the endpoints of e.

Proposition 7 (Wait-for-edge preservation). *Let $s \xrightarrow{a} t$ be a transition of composite component $B = \gamma(B_1, \ldots, B_n)$, and let e be a wait-for edge that is not B_i-incident, for every $B_i \in C_a$. Then $e \in W_B(s)$ iff $e \in W_B(t)$.*

Proof sketch. Components not involved in the execution of a do not change state along $s \xrightarrow{a} t$. Hence the endpoint of e that is a component has the same state in s as in t. The proposition then follows from Definition 12.

We show, by induction on the length of finite exeuctions, that every reachable state is supercycle-free. Assume that every initial state is supercycle-free, for the base case. Assuming $\mathcal{DFC}(a)$ for all $a \in \gamma$ provides, by the above discussion, the induction step.

Theorem 1 (Deadlock-freedom). *If (1) for all $s_0 \in Q_0$, $W_B(s_0)$ is supercycle-free, and (2) for all interactions a of B (i.e., $a \in \gamma$), $\mathcal{DFC}(a)$ holds, then for every reachable state u of (B, Q_0): $W_B(u)$ is supercycle-free.*

Proof. We only need show the induction step: for every reachable transition $s \xrightarrow{a} t$, $W_B(s)$ is supercycle-free implies that $W_B(t)$ is supercycle-free. We establish the contrapositive: if $W_B(t)$ contains a supercycle, then so does $W_B(s)$.

Let SC be a supercycle in $W_B(t)$, and let SC' be SC with all nodes of finite in-depth removed. SC' is a supercycle in $W_B(t)$ by Proposition 6. Let e be an arbitrary edge in SC'. Hence $e \in W_B(t)$. Also, both nodes of e have infinite in-depth (by construction of SC') and infinite out-depth (by Proposition 4) in $W_B(t)$. Let B_i be an arbitrary component in C_a. By $\mathcal{DFC}(a)$, B_i has finite in-depth or finite out-depth in $W_B(t)$: $\mathit{in\_depth}_B(B_i, t) < \omega \vee \mathit{out\_depth}_B(B_i, t) < \omega$. Hence e is not B_i-incident. So, $e \in W_B(s)$, by Proposition 7. Hence $SC' \subseteq W_B(s)$, and so $W_B(s)$ contains a supercycle.

5 A Local Condition for Deadlock Freedom

Evaluating $\mathcal{DFC}(a)$ requires checking all reachable transitions of (B, Q_0), which is subject to state-explosion. We need a condition which implies $\mathcal{DFC}(a)$ and can be checked efficiently. Observe that if $in\_depth_B(B_i, t) < \omega \vee out\_depth_B(B_i, t) < \omega$, then there is some finite ℓ such that $in\_depth_B(B_i, t) = \ell \vee out\_depth_B(B_i, t) = \ell$. This can be verified in a subsystem whose size depends on ℓ, as follows.

Definition 17 (Structure Graph G_B, G_i^ℓ, G_a^ℓ). *The structure graph G_B of composite component $B = \gamma(B_1, \ldots, B_n)$ is a bipartite graph whose nodes are the $B_1, \ldots, B_n$ and all the $a \in \gamma$. There is an edge between B_i and interaction a iff B_i participates in a, i.e., $B_i \in C_a$. Define the* distance *between two nodes to be the number of edges in a shortest path between them. Let G_i^ℓ (G_a^ℓ respectively) be the subgraph of G_B that contains B_i (a respectively) and all nodes of G_B that have a distance to B_i (a respectively) less than or equal to ℓ.*

Then $in\_depth_B(B_i, t) = \ell \vee out\_depth_B(B_i, t) = \ell$ can be verified in the wait-for-graph of $G_i^{\ell+1}$, since we verify either that all wait-for-paths ending in B_i have length $\leq \ell$, or that all wait-for-paths starting in B_i have length $\leq \ell$. These conditions can be checked in $G_i^{\ell+1}$, since $G_i^{\ell+1}$ contains every node in a wait-for-path of length $\ell + 1$ or less and which starts or ends in B_i. Since $G_i^{\ell+1} \subseteq G_a^{\ell+2}$ for $B_i \in C_a$, we use $G_a^{\ell+2}$ instead of the set of subsystems $\{G_i^{\ell+1} : B_i \in C_a\}$. We leave analysis of the tradeoff between using one larger system ($G_a^{\ell+2}$) versus several smaller ones ($G_i^{\ell+1}$) to another paper. Define D_a^ℓ, the *deadlock-checking subsystem for interaction a and depth ℓ*, to be the subsystem of (B, Q_0) based on $G_a^{\ell+2}$.

Definition 18 ($\mathcal{LDFC}(a, \ell)$). *Let $s_a \xrightarrow{a} t_a$ be a reachable transition of D_a^ℓ. Then, in t_a, the following holds. For every component B_i of C_a: either B_i has in-depth at most ℓ, or out-depth at most ℓ, in $W_{D_a^\ell}(t_a)$. Formally,*

$$\forall B_i \in C_a : in\_depth_{D_a^\ell}(B_i, t_a) \leq \ell \vee out\_depth_{D_a^\ell}(B_i, t_a) \leq \ell.$$

To infer deadlock-freedom in (B, Q_0) by checking $\mathcal{LDFC}(a, \ell)$, we show that wait-for behavior in B "projects down" to any subcomponent B', and that wait-for behavior in B' "projects up" to B.

Proposition 8 (Wait-for-edge projection). *Let (B', Q'_0) be a subsystem of (B, Q_0). Let s be a state of (B, Q_0), and $s' = s{\upharpoonright}B'$. Let a be an interaction of (B', Q'_0), and $B_i \in C_a$ an atomic component of B'. Then (1) $a \to B_i \in W_B(s)$ iff $a \to B_i \in W_{B'}(s')$, and (2) $B_i \to a \in W_B(s)$ iff $B_i \to a \in W_{B'}(s')$.*

Proof sketch. Since $s' = s{\upharpoonright}B'$, all port enablement conditions of components in B' have the same value in s and in s'. The proposition then follows by straightforward application of Definition 12.

Since wait-for-edges project up and down, it follows that wait-for-paths project up and down, provided that the subsystem contains the entire wait-for-path.

Proposition 9 (In-projection, Out-projection). *Let $\ell \geq 0$, let B_i be an atomic component of B, and let (B', Q_0') be a subsystem of (B, Q_0) which is based on a superset of $G_i^{\ell+1}$. Let s be a state of (B, Q_0), and $s' = s{\upharpoonright}B'$. Then (1) $in\_depth_B(B_i, s) \leq \ell$ iff $in\_depth_{B'}(B_i, s') \leq \ell$, and (2) $out\_depth_B(B_i, s) \leq \ell$ iff $out\_depth_{B'}(B_i, s') \leq \ell$.*

Proof sketch. Follows from Defintion 15, Proposition 8, and the observation that $W_{B'}(s')$ contains all wait-for-paths of length $\leq \ell$ that start or end in B_i.

We now show that $\mathcal{LDFC}(a, \ell)$ implies $\mathcal{DFC}(a)$, which in turn implies deadlock-freedom.

Lemma 1. *Let a be an interaction of B, i.e., $a \in \gamma$. If $\mathcal{LDFC}(a, \ell)$ holds for some finite $\ell \geq 0$, then $\mathcal{DFC}(a)$ holds.*

Proof sketch. Let $s \xrightarrow{a} t$ be a reachable transition of (B, Q_0) and let $s_a = s{\upharpoonright}D_a^\ell$, $t_a = t{\upharpoonright}D_a^\ell$. Then $s_a \xrightarrow{a} t_a$ is a reachable transition of D_a^ℓ by Corollary 1. By $\mathcal{LDFC}(a, \ell)$, $in\_depth_{D_a^\ell}(B_i, t_a) \leq \ell \vee out\_depth_{D_a^\ell}(B_i, t_a) \leq \ell$. Hence by Proposition 9, $in\_depth_B(B_i, t) \leq \ell \vee out\_depth_B(B_i, t) \leq \ell$. So $in\_depth_B(B_i, t) < \omega \vee out\_depth_B(B_i, t) < \omega$. Hence $\mathcal{DFC}(a)$ holds.

Theorem 2 (Deadlock-freedom). *If (1) for all $s_0 \in Q_0$, $W_B(s_0)$ is supercycle-free, and (2) for all interactions a of B ($a \in \gamma$), $\mathcal{LDFC}(a, \ell)$ holds for some $\ell \geq 0$, then for every reachable state u of (B, Q_0): $W_B(u)$ is supercycle-free.*

Proof sketch. Immediate from Lemma 1 and Theorem 1.

6 Implementation and Experimentation

LDFC-BIP, ($\sim$ 1500 LOC Java) implements our method for finite-state BIP-systems. Pseudocode for LDFC-BIP is shown in Figure 4. checkDF(B, Q_0) iterates over each interaction a of (B, Q_0), and checks $(\exists \ell \geq 0 : \mathcal{LDFC}(a, \ell))$ by starting with $\ell = 0$ and incrementing ℓ until either $\mathcal{LDFC}(a, \ell)$ is found to hold, or D_a^ℓ has become the entire system and $\mathcal{LDFC}(a, \ell)$ does not hold. In the latter case, $\mathcal{LDFC}(a, \ell)$ does not hold for any finite ℓ, and, in practice, computation would halt before D_a^ℓ had become the entire system, due to exhaustion of resources.

locLDFC(a, ℓ) checks $\mathcal{LDFC}(a, \ell)$ by examining every reachable transition that executes a, and checking that the final state satisfies Definition 18.

The running time of our implementation is $O(\Sigma_{a \in \gamma} |D_a^{\ell_a}|)$, where ℓ_a is the smallest value of ℓ for which $\mathcal{LDFC}(a, \ell)$ holds, and where $|D_a^{\ell_a}|$ denotes the size of the transition system of $D_a^{\ell_a}$.

6.1 Experiment: Dining Philosophers

We consider n philosophers in a cycle, based on the components of Figure 1. Figure 5(a) provides experimental results. The x axis gives the number n of philosophers (and also the number of forks), and the y axis gives the verification time (in milliseconds). We verified that $\mathcal{LDFC}(a, \ell)$ holds for $\ell = 1$ and all interactions a. Hence dining philosophers is deadlock-free. We increase n and plot the

```
checkDF$(B, Q_0)$, where $B \triangleq \gamma(B_1, \dots, B_n)$
1.  forall interactions $a \in \gamma$
2.      //check $(\exists \ell \geq 0 : \mathcal{LDFC}(a, \ell))$
3.      $\ell \leftarrow 0$;                                        //start with $\ell = 0$
4.      while (true)
5.          if (locLDFC$(a, \ell)$ = true) break endif;     //success, so go on to next $a$
6.          if $(D_a^\ell = \gamma(B_1, \dots, B_n))$ return(false) endif;
7.          $\ell \leftarrow \ell + 1$          //increment $\ell$ until success or intractable or failure
8.      endwhile
9.  endfor;
10. return(true)                        //return true if check succeeds for all $a \in \gamma$

locLDFC$(a, \ell)$
1.  forall reachable transitions $s_a \xrightarrow{a} t_a$ of $D_a^\ell$
2.      if $(\neg(\forall B_i \in C_a : \mathit{in\_depth}_{D_a^\ell}(B_i, t_a) = \ell \vee \mathit{out\_depth}_{D_a^\ell}(B_i, t_a) = \ell))$
3.          return(false)                                   //check Definition 18
4.  endfor;
5.  return(true)                    //return true if check succeeds for all transitions
```

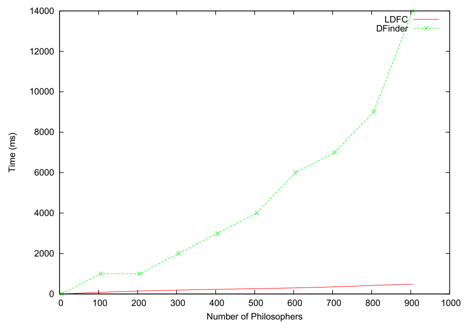
Fig. 4. Pseudocode for the implementation of our method

verification time for both LDFC-BIP and D-Finder 2 [8]. D-Finder 2 implements a compositional and incremental method for the verification of BIP-systems. D-Finder (the precursor of D-Finder 2) has been compared favorably with NuSmv and SPIN, outperforming both NuSmv and SPIN on dining philosophers, and outperforming NuSmv on the gas station example [7], treated next. Our results show that LDFC-BIP has a linear increase of computation time with the system size (n), and so outperforms D-Finder 2.

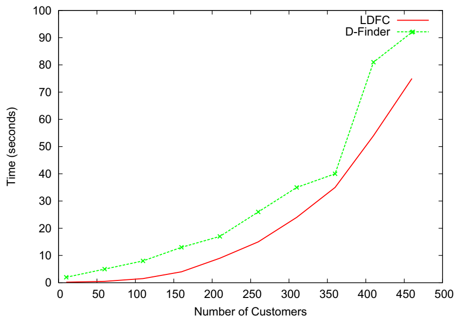
6.2 Experiment: Gas Station

A gas station [13] consists of an operator, a set of pumps, and a set of customers. Before using a pump, a customer has to prepay. Then the customer uses the pump, collects his change and starts a new transaction. Before being used by a customer, a pump has to be activated by the operator. When a pump is shut off, it can be re-activated for the next operation.

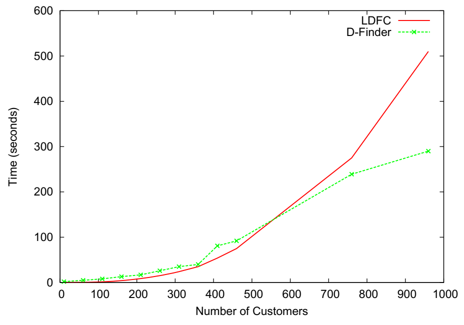
We verified $\mathcal{LDFC}(a, \ell)$ for $\ell = 2$ and all interactions a. Hence gas station is deadlock-free. Figures 5(b), 5(c), and 5(d) present the verification times using LDFC-BIP and D-Finder 2. We consider a system with 3 pumps and variable number of customers. In these figures, the x axis gives the number n of customers, and the y axis gives the verification time (in seconds). D-Finder 2 suffers state-explosion at $n = 1800$, because we consider only three pumps, and so the incremental method used by D-Finder 2 deteriorates. LDFC-BIP outperforms D-Finder 2 as the number of customers increases.



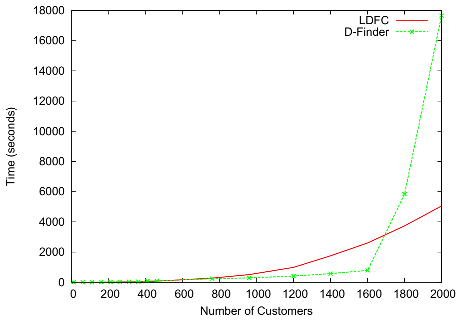
(a) Dining philosophers benchmark.



(b) Gas station benchmark 1.



(c) Gas station benchmark 2.



(d) Gas station benchmark 3.

Fig. 5. Benchmarks generated by our experiments

7 Discussion, Related Work, and Further Work

Related Work. The notions of wait-for-graph and supercycle [3, 4] were initially defined for a shared memory program $P = P_1 \parallel \cdots \parallel P_K$ in *pairwise normal form*: a binary symmettric relation I specifies the directly interacting pairs ("neighbors") $\{P_i, P_j\}$. If P_i has neighbors P_j and P_k, then the code in P_i that interacts with P_j is expressed separately from the code in P_i that interacts with P_k. These synchronization codes are executed synchronously and atomically, so the grain of atomicity is proportional to the degree of I. Attie and Chockler [3] give two polynomial time methods for deadlock freedom. The first checks subsystems consisting of three processes. The second computes the wait-for-graphs of all pair subsystems $P_i \parallel P_j$, and takes their union, for all pairs and all reachable states of each pair. The first method considers only wait-for-paths of length ≤ 2. The second method is prone to false negatives, because wait-for edges generated by different states are all merged together, which can result in spurious supercycles.

Gössler and Sifakis [12] use a BIP-like formalism, Interaction Models. They present a criterion for global deadlock freedom, based on an and-or graph with components and constraints as the two sets of nodes. A constraint gives the condition under which a component is blocked. Edges are labeled with conjuncts of the constraints. Deadlock freedom is checked by traversing every cycle, taking the conjunction of all the conditions labeling its edges, and verifying that this conjunction is always false, i.e., verifying the absence of cyclical blocking. No complexity bounds are given. Martens and Majster-Cederbaum [14] present a polynomial time checkable deadlock freedom condition based on structural restrictions: "the communication structure between the components is given by a tree." This restriction allows them to analyze only pair systems. Brookes and Roscoe [11] provide criteria for deadlock freedom of CSP programs based on structural and behavioral restrictions combined with analysis of pair systems. No implementation, or complexity bounds, are given. Aldini and Bernardo [1] use a formalism based on process algebra. They check deadlock by analysing cycles in the connections between software components, and claim scalability, but no complexity bounds are given.

We compared our implementation LDFC-BIP to D-Finder 2 [8]. D-Finder 2 computes a finite-state abstraction for each component, which it uses to compute a global invariant I. It then checks if I implies deadlock freedom. Unlike LDFC-BIP, D-Finder 2 handles infinite state systems. However, LDFC-BIP had superior running time for dining philosophers and gas station (both finite-state).

All the above methods verify global (and not local) deadlock-freedom. Our method verifies both. Also, our approach makes no structural restriction at all on the system being checked for deadlock.

Discussion. Our approach has the following advantages:

Local and Global Deadlock. Our method shows that no subset of processes can be deadlocked, i.e., absence of both local and global deadlock.

Check Works for Realistic Formalism. By applying the approach to BIP, we provide an efficient deadlock-freedom check within a formalism from which efficient distributed implementations can be generated [9].

Locality. If a component B_i is modified, or is added to an existing system, then $\mathcal{LDFC}(a, \ell)$ only has to be re-checked for B_i and components within distance ℓ of B_i. A condition whose evaluation considers the entire system at once, e.g., [1, 8, 12] would have to be re-checked for the entire system.

Easily Parallelizable. Since the checking of each subsystem D_a^ℓ is independent of the others, the checks can be carried out in parallel. Hence our method can be easily parallelized and distributed, for speedup, if needed. Alternatively, performing the checks sequentially minimizes the amount of memory needed.

Framework Aspect. Supercycles and in/out-depth provide a *framework* for deadlock-freedom. Conditions more general and/or discriminating than the one presented here should be devisable in this framework. This is a topic for future work.

Further Work. Our implementation uses explicit state enumeration. Using BDD's may improve the running time when $\mathcal{LDFC}(a, \ell)$ holds only for large ℓ. An enabled port p enables all interactions containing p. Deadlock-freedom conditions based on ports could exploit this interdepence among interaction enablement. Our implementation should produce *counterexamples* when a system fails to satisfy $\mathcal{LDFC}(a, \ell)$. *Design rules* for ensuring $\mathcal{LDFC}(a, \ell)$ will help users to produce deadlock-free systems, and also to interpret counterexamples. A *fault* may create a deadlock, i.e., a supercycle, by creating wait-for-edges that would not normally arise. Tolerating a fault that creates up to f such spurious wait-for-edges requires that there do not arise during normal (fault-free) operation subgraphs of $W_B(s)$ that can be made into a supercycle by adding f edges. We will investigate criteria for preventing formation of such subgraphs. Methods for evaluating $\mathcal{LDFC}(a, \ell)$ on *infinite state* systems will be devised, e.g.,, by extracting proof obligations and verifying using SMT solvers. We will extend our method to *Dynamic BIP*, [10], where participants can add and remove interactions at run time.

References

1. Aldini, A., Bernardo, M.: A General Approach to Deadlock Freedom Verification for Software Architectures. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 658–677. Springer, Heidelberg (2003)
2. Attie, P.C.: Synthesis of large concurrent programs via pairwise composition. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 130–145. Springer, Heidelberg (1999)
3. Attie, P.C., Chockler, H.: Efficiently verifiable conditions for deadlock-freedom of large concurrent programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 465–481. Springer, Heidelberg (2005)
4. Attie, P.C., Allen Emerson, E.: Synthesis of Concurrent Systems with Many Similar Processes. TOPLAS 20(1), 51–115 (1998)
5. Attie, P.C., Francez, N., Grumberg, O.: Fairness and Hyperfairness in Multiparty Interactions. Distributed Computing 6, 245–254 (1993)
6. Basu, A., Bozga, M., Sifakis, J.: Modeling Heterogeneous Real-time Components in BIP. In: SEFM, pp. 3–12 (September 2006)
7. Bensalem, S., Bozga, M., Nguyen, T.H., Sifakis, J.: Compositional verification for component-based systems and application. IET Software 4(3), 181–193 (2010)
8. Bensalem, S., Griesmayer, A., Legay, A., Nguyen, T.-H., Sifakis, J., Yan, R.: D-finder 2: Towards efficient correctness of incremental design. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 453–458. Springer, Heidelberg (2011)
9. Bonakdarpour, B., Bozga, M., Jaber, M., Quilbeuf, J., Sifakis, J.: From High-level Component-based Models to Distributed Implementations. In: EMSOFT, pp. 209–218 (2010)
10. Bozga, M., Jaber, M., Maris, N., Sifakis, J.: Modeling Dynamic Architectures Using Dy-BIP. In: Gschwind, T., De Paoli, F., Gruhn, V., Book, M. (eds.) SC 2012. LNCS, vol. 7306, pp. 1–16. Springer, Heidelberg (2012)

11. Brookes, S.D., Roscoe, A.W.: Deadlock analysis in networks of communicating processes. Distributed Computing 4, 209–230 (1991)
12. Göler, G., Sifakis, J.: Component-based construction of deadlock-free systems. In: Pandya, P.K., Radhakrishnan, J. (eds.) FSTTCS 2003. LNCS, vol. 2914, pp. 420–433. Springer, Heidelberg (2003)
13. Heimbold, D., Luckham, D.: Debugging Ada tasking programs. IEEE Software 2(2), 47–57 (1985)
14. Martens, M., Majster-Cederbaum, M.: Deadlock-freedom in component systems with architectural constraints. FMSD 41, 129–177 (2012)
15. Papadimitriou, C.H.: Computational complexity. Addison-Wesley (1994)

Bounded Model Checking of Graph Transformation Systems via SMT Solving

Tobias Isenberg, Dominik Steenken, and Heike Wehrheim

Universität Paderborn
Institut für Informatik
33098 Paderborn, Germany
{isenberg,dominik,wehrheim}@mail.upb.de

Abstract. Bounded model checking (BMC) complements classical model checking by an efficient technique for checking error-freedom of bounded system paths. Usually, BMC approaches reduce the verification problem to propositional satisfiability. With the recent advances in SAT solving, this has proven to be a fast analysis.

In this paper we develop a bounded model checking technique for graph transformation systems. Graph transformation systems (GTSs) provide an intuitive, visual way of specifying system models and their structural changes. An analysis of such models – however – remains difficult since GTSs often give rise to infinite state spaces. In our BMC technique we use *first-order* instead of propositional logic for encoding complex graph structures and rules. Today's off-the-shelf SMT solvers can then readily be employed for satisfiability solving. The encoding heavily employs the concept of *uninterpreted function* symbols for representing edge labels. We have proven soundness of the encoding and report on experiments with different case studies.

Keywords: verification, graph transformation systems, bounded model checking, satisfiablility modulo theories.

1 Introduction

Graph transformation systems (GTSs) are an intuitive and powerful way of modeling dynamic systems. They describe the states of a system as graphs and state changes as graph transformation rules. This makes them particularly suitable for modeling structural aspects of systems. GTSs have been applied in diverse areas, such as model transformation and refactorings [9], the modeling of dynamic self-adaptive systems [6] or web services [14]. Some of these application areas are safety critical, raising the need for formal verification of such systems. When modeling these systems using GTSs, the analysis often suffers from the state space explosion problem. A number of approaches aim at fighting this problem, most often using *abstraction* techniques [25,2].

Bounded model checking [8] is a technique avoiding the state space explosion problem by focusing on bounded paths. This technique is incomplete, and thus primarily used for finding errors, not for correctness proofs. This idea has been adapted for the verification of GTSs [17,3], however, few of the existing solutions directly utilizes the progress of SAT/SMT solvers seen in the last years. In particular, none of the verification techniques for GTSs have proposed to reduce the complete analysis directly to satisfiability checking. An encoding of GTS in logical formulae is only used in [20], which however gives a SAT encoding and uses the result to guide the state space exploration of a (non-SAT) analysis tool for GTS.

In this paper we present a bounded model checking (BMC) technique for GTSs using satisfiability checking of quantifier-free first-order logic with uninterpreted functions (QF_UF). We transform the BMC instance over GTSs into a satisfiability modulo theories (SMT) instance encoding only the initial graph and the transformation rules, as well as the property to be checked.

In the present paper, the property will – for simplicity – always be a (forbidden) graph; thus we just aim at checking reachability of error states. Our approach, detailed in [16], does however cover more general properties specified in LTL. The generated SMT formula describes bounded-length paths of the GTS that exhibit the forbidden graph. The formula is given to an off-the-shelf SMT solver which checks for satisfiability. A satisfying interpretation represents a violating path and is used to display it.

We have implemented our approach and tested it with a number of SMT solvers (MathSAT[12], SMTInterpol[11], veriT[10], Z3[21]). The performance of the approach heavily depends on the structure of rules (e.g., number of nodes, NACs), and only to a minor extent on the solver. We exemplify our technique on the car platooning case study of the Transformation Tool Contest 2010 [1].

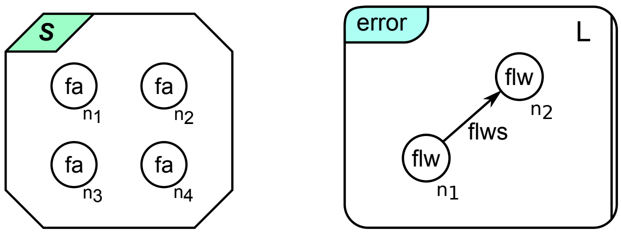
2 Background

We start by defining *Graph Transformation Systems* (GTSs) and the reachability problem of a forbidden pattern in this context, while introducing the running example. The definitions are similar to those in [13,15,22].

Definition 1 (Graph). *Let L_E be a set of edge labels and L_N be a set of node labels. A* graph $G = (N_G, E_G, U_G)$ *over L_E and L_N consists of a set N_G of nodes, a set $E_G \subseteq N_G \times L_E \times N_G$ of labeled edges and a set $U_G \subseteq L_N \times N_G$ of unary edges labeling the nodes.*

A graph is a set of nodes with edges between pairs of them and on individual nodes. These edges are labeled over L_E and L_N respectively.

For illustration purposes, consider the car platooning GTS specified by Backes and Reineke [1], modeling a protocol for cars to build a platoon, i.e., a connected entity. Initially the cars are unconnected and in the state *free agent*(*fa*). During the process of connecting there are several intermediate states, as described by the protocol. However, the final states of a car are *leader*(*ld*) and *follower*(*flw*). A platoon is structured such that there are a number of *follower*s following one *leader*.



(a) Initial graph S (b) Forbidden pattern

Fig. 1. Part of the car platooning GTS [1]

In our example this protocol is modeled as a GTS with an initial graph containing four nodes labeled *fa*, modeling cars in state *free agent*(see Figure 1a). Note that we model node labels as "unary" edges, i.e., edges with a target but without a source. Additionally, a GTS has rules that describe possible changes and under which conditions these can be applied. For these transformation rules and their applicability consider the following definitions.

Definition 2. *A* total graph morphism $f : G \to G'$ *from graph* G *to graph* G' *is a tuple* $f = \langle f_N, f_E, f_U \rangle$ *of total functions* $f_N : N_G \to N_{G'}$, $f_E : E_G \to E_{G'}$ *and* $f_U : U_G \to U_{G'}$ *satisfying the following conditions:*

$$\begin{aligned} f_E(n_1, l, n_2) &= (f_N(n_1), l, f_N(n_2)) \qquad && \forall (n_1, l, n_2) \in E_G \\ f_U(l, n_1) &= (l, f_N(n_1)) && \forall (l, n_1) \in U_G \end{aligned}$$

A partial graph morphism from graph G to graph G' is a total graph morphism of a subgraph S of G to G'. A total graph morphism specifies a mapping that maintains the structure. Intuitively speaking, such a morphism exists, if a subgraph similar to G can be found in G'. This construct is fundamental for applying graph transformation rules. Such a rule consists of a left hand side graph L and a right hand side graph R, related by a partial graph morphism.

Definition 3 (Transformation Rule and Match). *Let* G *be a graph, called host graph. A* graph transformation rule $r = \langle L, R \rangle$*, described by a partial graph morphism* $r : L \to R$*, is applicable to* G*, if and only if there exists a total, injective graph morphism* $m : L \to G$*, called a* match.

For simplicity, in the following we assume $r = id_{L \cap R}$ (i.e., L and R need not be disjoint, e.g., see the nodes in Figure 2). Intuitively speaking, a match specifies at which location in the graph a rule can be applied. This application itself is done obeying the well known SPO-semantics [13] explained below.

Let G, r and m be as in Def. 3. We define $N_{del} := N_L \setminus N_R$ and N_{add} as a disjoint copy of $N_R \setminus N_L$. By setting $m_N := m \cup id_{N_{add}}$ we extend m onto N_{add}. The following sets are useful in defining the effect of a rule application.

$$E_{del} := \{(m_N(n_1), l, m_N(n_2)) \mid (n_1, l, n_2) \in E_L \setminus E_R \vee n_1 \in N_{del} \vee n_2 \in N_{del}\}$$
$$E_{add} := \{(m_N(n_1), l, m_N(n_2)) \mid (n_1, l, n_2) \in E_R \setminus E_L\}$$

Corresponding sets U_{del} and U_{add} for unary edges are defined analogously. Each of these sets contains the objects deleted / added through the rule application.

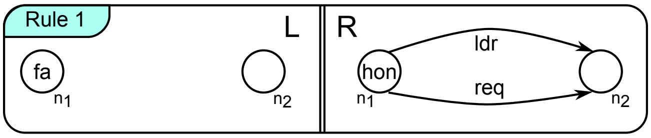


Fig. 2. Rule 1 of the car platooning GTS [1]

This gives rise to the following definition.

Definition 4 (Rule Application). *Given a host graph G, a graph transformation rule $r = \langle L, R\rangle$ and a match $m : L \to G$. The* application of *r to graph G via the match m, written $G \overset{r,m}{\Longrightarrow} H$, creates a result graph H specified by $N_H := N_G \setminus m_N(N_{del}) \cup N_{add}$, $E_H := E_G \setminus E_{del} \cup E_{add}$, and $U_H := U_G \setminus U_{del} \cup U_{add}$.*

The resulting graph is constructed such that the images (w.r.t. the match) of the nodes and edges of the LHS, which are not part of the RHS have to be deleted. Afterwards, new nodes and edges are added for the ones contained in the RHS and not in the LHS. All dangling edges, i.e., edges that are left without a source- or target node after application, are deleted.

Definition 5 (GTS). *A* graph transformation system *$\mathcal{G} = (S, P)$ consists of a start graph S and a set $P = \{r_1, ..., r_n\}$ of graph transformation rules.*

The car platooning GTS [1] consists of 14 rules. For simplicity we will only present the encoding of two of them (*Rules 1 and 13*) illustrated in the next section (see Figures 2,3). The LHS of *Rule 1* requires two distinct nodes in the host graph, one of which is labeled *fa*(*free agent*) and changes its state to *hon*(*hand over nothing*). Additionally, two edges are added as shown in Fig. 2. This rule models the initialization of a connection started by a *free agent* car.

For *Rule 13*, we will need an extension of the above formalism. We need to allow Negative Application Conditions (NACs) within the rules [15]. A NAC constrains the application of a rule, s.t. it is only applicable if the structures described by the NAC are not present in the host graph.

Definition 6 (NAC). *A* negative application condition *for a graph transformation rule $r = \langle L, R\rangle$ is a set C of total graph morphisms with domain L.*

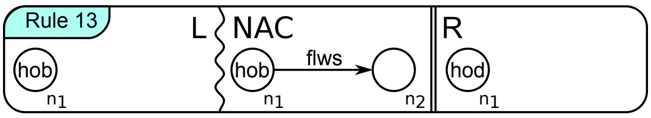


Fig. 3. Rule 13 of the car platooning GTS [1]

Let $l : L \to \hat{L}$ be such a morphism and let m be a match of r to G. The application of r with match m to the host graph G is allowed w.r.t. l, if and only if no match $n : \hat{L} \to G$ exists with $n \circ l = m$ (concatenation operator $\circ$). A NAC allows the application of r with match m to the host graph G, if and only if the application is allowed w.r.t. all total graph morphisms of the NAC.

For simplicity, in the following we assume l to be an embedding, i.e., $L \subset \hat{L}$. Rule 13 (Figure 3) contains a NAC. It states that the match of the node n_1 must not have a *flws*-labeled edge to any other node. When two platoons join, one leader hands over all his followers to the other leader. While having at least one follower left, he is not allowed to proceed to the next step of the protocol, but must finish the handover first. This is modeled by the NAC in Rule 13. Thus, car n_1 must not have a follower, when changing its state from *hand over back* to *hand over done*. With this understanding of the application of rules, we define the paths of a GTS and the reachability problem.

Definition 7. *Given a GTS $\mathcal{G} = (S, P)$. A state of the GTS is a graph. Thus, a* path *$G_0, G_1, \ldots$ of $\mathcal{G}$ is a finite or infinite sequence of graphs starting with the initial graph ($G_0 = S$) such that we have $G_i \overset{r_i,m_i}{\Longrightarrow} G_{i+1}$ for all consecutive graphs G_i,G_{i+1}. The* state space *of a GTS is the union of all of its paths, joined at isomorphic graphs.*

While usually, GTSs do not distinguish between isomorphic graphs, our encoding does not have that capability. Thus, each isomorphic instance of a graph is explored separately. In the following we state the reachability problem.

Definition 8. *Given a GTS $\mathcal{G}$ and a graph F, called the forbidden pattern. The pattern is* reachable*, if there exists a finite path $G_0, \ldots, G_k$ of $\mathcal{G}$, such that F matches G_k.*

Intuitively speaking, reachability means the existence of a path containing a graph in which the forbidden pattern can be found. Within the car platooning protocol [1], no follower should ever follow a follower. This can be expressed as a forbidden pattern (see Figure 1b). If that pattern would be reachable, the model of the protocol would not meet the desired behavior.

We propose a BMC technique to check reachability of such a forbidden graph. Our approach directly encodes the initial graph, the transformation rules and the forbidden pattern as a satisfiability modulo theories (SMT) instance. SMT describes the problem of searching for a satisfying interpretation of a first-order logic formula with a background theory. Intuitively, a theory can be seen as constraints on the possible interpretations, as it describes a priori interpretations of functions. Our formulae use the theory of uninterpreted functions with equality, i.e., only the equality sign has an a priori interpretation. This approach relocates the computational complexity of exploring the bounded state space into solving an SMT formula, which can be a reasonable trade-off, as current research continuously improves efficiency of SMT solvers (SMT-COMP [4]).

The technique presented here is subject to several restrictions, which we summarize here for clarity.

1. we only allow simple, labeled graphs (no multigraphs, no hypergraphs)
2. we disallow node merging
3. we disallow non-injective matches
4. we only support reachability analysis

Restrictions 3 and 4 are removed in our full technique, detailed in [16]. In the following, we will only give the definitions required for the restricted version.

3 Encoding of GTSs in First-Order Logic

In the following we describe our BMC technique, which encodes bounded paths of a GTS (S, P) as an SMT formula to check the reachability of a forbidden pattern in the bounded state space. The concept of BMC via SAT-solving was originally introduced in 1999 [8]. In hardware verification it is a natural choice to represent circuits as boolean formulas. This idea was then extended to check error freedom of a bounded subset of the state space. The state space is only examined up to a predefined number k of steps, called the bound. The initial state, k transitions and the error are encoded as a boolean formula, that is satisfiable if the error exists within the bounded state space.

Our approach likewise encodes the start graph, k transitions and a forbidden pattern (or LTL formula). A transition in this context denotes the application of one of the graph transformation rules of the GTS.

Let $[\![S]\!]$ be the encoding of the initial graph $S = G_0$ and $[\![T]\!]_1$ to $[\![T]\!]_k$ be the encodings of the k transition steps. The complete formula is defined as $[\![S]\!] \wedge [\![T]\!]_1 \wedge ... \wedge [\![T]\!]_k \wedge [\![F]\!]$ with $[\![F]\!]$ being the encoding of the forbidden pattern.

Below we describe the process in detail. Let L_E be the set of all edge labels occurring in the initial graph, the rules or the forbidden pattern and let L_N be the set of all node labels, respectively. For representing all edges with label $l \in L_E$ of a graph $G_i (i \in \{0, ..., k\})$ along the path $G_0, ..., G_k$ we use binary predicates l_i. For tagging the nodes with label $l \in L_N$ we use unary predicates, respectively. Let $\mathcal{U}$ be the universe consisting of the nodes of the initial graph. Unary predicates have domain $\mathcal{U}$, binary ones have domain $\mathcal{U} \times \mathcal{U}$.

Definition 9. *Let* $L_{E,i} = \{l_i | l \in L_E\}$ *and* $L_{N,i} = \{l_i | l \in L_N\}$ *be the sets of binary and unary predicates with index* i*. An* i-interpretation $\mathcal{I}_i$ *assigns a total function with image* $true$ *or* $false$ *to every such predicate. Formally,* $\mathcal{I}_i$:

- $L_{E,i} \to \{\mathcal{U} \times \mathcal{U} \to \{true, false\}\}$
- $L_{N,i} \to \{\mathcal{U} \to \{true, false\}\}$.

*An i-*interpretation *$\mathcal{I}_i$ represents a graph G_i (written $\mathcal{I}_i(G_i)$), iff:*

- $\forall l \in L_E \, \forall n_1, n_2 \in N_{G_i} : \;\; \mathcal{I}_i(l_i)(n_1, n_2) = true \Leftrightarrow (n_1, l, n_2) \in E_{G_i}$
- $\forall l \in L_N \, \forall n \in N_{G_i} : \;\; \mathcal{I}_i(l_i)(n) = true \Leftrightarrow (l, n) \in U_{G_i}$.

This idea of how a satisfying interpretation represents a graph is used throughout the complete encoding process.

Encoding of the Initial Graph. Given the initial graph $S = G_0 = (N_{G_0}, E_{G_0}, U_{G_0})$, we encode each edge $(n_1, l, n_2) \in E_{G_0}$ with a positive literal $l_0\,(n_1, n_2)$ and each nonexistent edge $(n_1, l, n_2) \in (\mathcal{U} \times L_E \times \mathcal{U}) \setminus E_{G_0}$ with a negated literal $\neg l_0\,(n_1, n_2)$. Unary edges are handled analogously. Furthermore, we include literals $\neg\,(n_i = n_j)$ for all pairings of distinct nodes n_i, n_j. This ensures that the graph itself and not a smaller graph it would be homomorphic to is encoded, and provides support for isolated nodes. These literals are combined via conjunction.

Example 1. Our example GTS has an initial graph as shown in Figure 1a and two transformation rules. These can be seen in Figures 2 and 3. The sets of labels in the example GTS are $L_E = \{\mathit{flws}, \mathit{ldr}, \mathit{req}\}$ and $L_N = \{\mathit{fa}, \mathit{hob}, \mathit{hod}, \mathit{hon}, \mathit{flw}\}$. $\mathcal{U}$ consists of the nodes from the start graph, called n_1 to n_4. The encoding of the initial graph is straightforward as there are no binary edges in the initial graph and the nodes are only labeled with *fa*:

$$[\![G_0]\!] = \bigwedge_{i=1}^{4} \mathit{fa}_0\,(n_i) \wedge \bigwedge_{\substack{l \in L_N \\ l \neq fa}} \bigwedge_{i=1}^{4} \neg l_0\,(n_i) \wedge \bigwedge_{\substack{i=1 \\ j=1}}^{4} \bigwedge_{l \in L_E} \neg l_0\,(n_i, n_j) \wedge \bigwedge_{\substack{i=1 \\ j=1 \\ i \neq j}}^{4} \neg\,(n_i = n_j)$$

The following lemma states the correctness of this encoding.

Lemma 1. *Given a graph G, $[\![G]\!]$ is satisfiable only with $\mathcal{I}_0(G)$ representing G.*

Encoding of a Transition. The i-th transition ($i \in \{1, ..., k\}$) can be caused by the application of any of the graph transformation rules to the host graph G_{i-1}. For a GTS with the set $P = \{r_1, ..., r_n\}$ of rules, the i-th transition is encoded as $[\![T]\!]_i = [\![r_1]\!]_i \vee ... \vee [\![r_n]\!]_i$ with $[\![r]\!]_i = [\![r]\!]_i^{Cond} \wedge [\![r]\!]_i^{App}$. Thus, we have to encode the applicability and the application of each rule for each transition step.

Encoding $[\![r]\!]_i^{Cond}$ of the Applicability of a Transformation Rule r. We first encode the check whether a rule is applicable to a host graph G_j (Note that $j = i - 1$ for the i-th transition). For each node $n \in N_L$ create a variable m_n^j, to which we want the SMT-Solver to assign a node from the universe $\mathcal{U}$. In general, SMT uses sorts with an unbounded number of members. Thus, we have to assure that only nodes out of $\mathcal{U}$ are used. We do this with the disjunction $(m_n^j = n_1) \vee ... \vee (m_n^j = n_2)$ for $\mathcal{U} = \{n_1, ..., n_2\}$. The assignment of the variables to the actual nodes in the universe represents the match. Since the match must be injective, for each two distinct nodes $n_1, n_2 \in N_L$ the literal $\neg\left(m^j_{n_1} = m^j_{n_2}\right)$ is added. In addition, each edge $(n_1, l, n_2) \in E_L$ of the LHS has to exist within the matched nodes of G_j. This is encoded by the literal $l_j\left(m^j_{n_1}, m^j_{n_2}\right)$. The analog holds true for the unary edges. Thus, we encode a unary edge $(l, n_1) \in U_L$ by a literal $l_j\left(m^j_{n_1}\right)$. All these literals are combined via conjunction.

Encoding of a NAC. If the transformation rule has Negative Application Conditions, these are encoded and added to the conjunction as well. Each total graph morphism of the NAC is encoded separately and combined afterwards by conjunction. Given such a morphism $L \to \hat{L}$, a variable m^j_n is created for each node $n \in \hat{L} \setminus L$. These new variables are bound by a universal quantifier and have to be distinct from each other and the previously generated variables of this rule. The quantified formula for an injective match is true if the negation of the conjunction of literals representing the binary and unary edges specified in $\hat{L} \setminus L$ holds ($l_j\left(m^j_{n_1}, m^j_{n_2}\right)$ for a given edge $(n_1, l, n_2) \in E_{\hat{L}} \setminus E_L$ and $l_j\left(m^j_n\right)$ for a unary edge $(l, n) \in U_{\hat{L}} \setminus U_L$). Within this negation we ensure an injective match with the tests that each newly created variable is disjoint from all other variables ($m^j_{n_1} = m^j_{n_2}$ for $n_1 \in \hat{L} \setminus L$ and $n_2 \in \hat{L}$).

Example 2. We continue the example and choose $k = 2$. We encode the injective applicability of both transformation rules (see Figures 2, 3) for the first transition step. (The encoding for the second transition step only differs by the indices.)

$$\begin{aligned}
[\![Rule1]\!]^{Cond}_1 =& fa_0\left(m^0_{n_1}\right) \wedge \neg\left(m^0_{n_1} = m^0_{n_2}\right) \wedge \bigvee_{n \in \mathcal{U}} \left(m^0_{n_1} = n\right) \wedge \bigvee_{n \in \mathcal{U}} \left(m^0_{n_2} = n\right) \\
[\![Rule13]\!]^{Cond}_1 =& hob_0\left(m^0_{n_1}\right) \wedge \bigvee_{n \in \mathcal{U}} \left(m^0_{n_1} = n\right) \wedge \\
& \forall m^0_{n_2} \in \mathcal{U} : \neg\left(\neg\left(m^0_{n_2} = m^0_{n_1}\right) \wedge \left(flws_0\left(m^0_{n_1}, m^0_{n_2}\right)\right)\right)
\end{aligned}$$

Note that, while we use quantifiers here to make the presentation more concise, these can be (and are) eliminated in the encoding by explicit enumeration of their range, thus making the resulting formula quantifier-free.

Lemma 2. *Given a graph transformation rule $r = \langle L, R\rangle$ and a host graph G_j. $[\![r]\!]^{Cond}_{j+1}$ is satisfiable with the interpretation $\mathcal{I}_j(G_j)$ representing G_j iff there exists a match $m : L \to G_j$ and the assignment of the variables is according to that match ($m^j_n = m_N(n)$).*

In the following we describe the encoding of the actual application of a graph transformation rule. Note, that we do not describe graph transformation rules with node creation/deletion here to keep illustration simple. Nevertheless, at the end of this section we briefly discuss these features.

Encoding $[\![r]\!]^{App}_i$ of the Application of a Transformation Rule r. Let graph G_j ($j = i - 1$) be the host graph as before. We use the previously created variables. Added binary edges $(n_1, l, n_2) \in E_R \setminus E_L$ have to exist in the resulting graph, encoded by a literal $l_{j+1}\left(m^j_{n_1}, m^j_{n_2}\right)$. Analogously, added unary edges are encoded by the corresponding positive unary literal. Similarly, deleted edges must not be present in the resulting graph and are encoded via the corresponding negated literals.

All other edges, which are neither deleted, nor added, exist in the resulting graph G_{j+1} if and only if they exist in the host graph G_j. This is represented by a universally quantified equivalence for each pair of two nodes from $\mathcal{U}$ from which we exclude the changed edges. Let $\{(n_1, l, n_2), ..., (n_3, l, n_4)\} \subseteq (E_R \setminus E_L \cup E_L \setminus E_R)$ be the set of changed edges with label l, which we need to exclude. For each label $l \in L_E$ we encode as follows. (The same holds true analogously for unchanged unary edges. All these parts are combined by conjunction.)

$$\begin{aligned}
\forall n_s, n_t \in \mathcal{U} :& (l_j\,(n_s, n_t) \Leftrightarrow l_{j+1}\,(n_s, n_t) && \text{same interpretation}\\
&\vee \left(\left(n_s = m^j_{n_1}\right) \wedge \left(n_t = m^j_{n_2}\right)\right) \vee ... && \text{except for all the}\\
&\vee \left(\left(n_s = m^j_{n_3}\right) \wedge \left(n_t = m^j_{n_4}\right)\right)). && \text{changed edges}
\end{aligned}$$

Example 3. We continue the example. As mentioned before, the encodings for the second transition step differs only by the indices.

$$\begin{aligned}
[\![Rule1]\!]^{App}_1 =& hon_1\left(m^0_{n_1}\right) \wedge ldr_1\left(m^0_{n_1}, m^0_{n_2}\right) \wedge req_1\left(m^0_{n_1}, m^0_{n_2}\right) \wedge \neg fa_1\left(m^0_{n_1}\right)\\
&\wedge \forall n_s, n_t \in \mathcal{U} : (flws_0\,(n_s, n_t) \Leftrightarrow flws_1\,(n_s, n_t))\\
&\wedge \forall n_s, n_t \in \mathcal{U} : ((ldr_0\,(n_s, n_t) \Leftrightarrow ldr_1\,(n_s, n_t)) \vee\\
&\qquad \left(\left(n_s = m^0_{n_1}\right) \wedge \left(n_t = m^0_{n_2}\right)\right))\\
&\wedge \forall n_s, n_t \in \mathcal{U} : ((req_0\,(n_s, n_t) \Leftrightarrow req_1\,(n_s, n_t)) \vee\\
&\qquad \left(\left(n_s = m^0_{n_1}\right) \wedge \left(n_t = m^0_{n_2}\right)\right))\\
&\wedge \forall n \in \mathcal{U} : \left((fa_0\,(n) \Leftrightarrow fa_1\,(n)) \vee \left(n = m^0_{n_1}\right)\right)\\
&\wedge \forall n \in \mathcal{U} : (hob_0\,(n) \Leftrightarrow hob_1\,(n))\\
&\wedge \forall n \in \mathcal{U} : (hod_0\,(n) \Leftrightarrow hod_1\,(n))\\
&\wedge \forall n \in \mathcal{U} : \left((hon_0\,(n) \Leftrightarrow hon_1\,(n)) \vee \left(n = m^0_{n_1}\right)\right)\\
&\wedge \forall n \in \mathcal{U} : (flw_0\,(n) \Leftrightarrow flw_1\,(n))
\end{aligned}$$

$$\begin{aligned}
[\![Rule13]\!]^{App}_1 =& \neg hob_1\left(m^0_{n_1}\right) \wedge hod_1\left(m^0_{n_1}\right)\\
&\wedge \forall n_s, n_t \in \mathcal{U} : (flws_0\,(n_s, n_t) \Leftrightarrow flws_1\,(n_s, n_t))\\
&\wedge \forall n_s, n_t \in \mathcal{U} : (ldr_0\,(n_s, n_t) \Leftrightarrow ldr_1\,(n_s, n_t))\\
&\wedge \forall n_s, n_t \in \mathcal{U} : (req_0\,(n_s, n_t) \Leftrightarrow req_1\,(n_s, n_t))\\
&\wedge \forall n \in \mathcal{U} : (fa_0\,(n) \Leftrightarrow fa_1\,(n))\\
&\wedge \forall n \in \mathcal{U} : \left((hob_0\,(n) \Leftrightarrow hob_1\,(n)) \vee \left(n = m^0_{n_1}\right)\right)\\
&\wedge \forall n \in \mathcal{U} : \left((hod_0\,(n) \Leftrightarrow hod_1\,(n)) \vee \left(n = m^0_{n_1}\right)\right)\\
&\wedge \forall n \in \mathcal{U} : (hon_0\,(n) \Leftrightarrow hon_1\,(n))\\
&\wedge \forall n \in \mathcal{U} : (flw_0\,(n) \Leftrightarrow flw_1\,(n))
\end{aligned}$$

Lemma 3. *Given a graph transformation rule $r = \langle L, R\rangle$ and a host graph G_j. $[\![r]\!]^{Cond}_{j+1} \wedge [\![r]\!]^{App}_{j+1}$ is satisfiable with an interpretation representing graphs G_j and G_{j+1} (i.e., $\mathcal{I}_j(G_j)$ and $\mathcal{I}_{j+1}(G_{j+1})$), iff $G_j \overset{r,m}{\Longrightarrow} G_{j+1}$ holds and the assignment of the variables is according to that match ($m^j_n = m_N(n)$).*

Encoding of the Forbidden Pattern. In order to check reachability of the forbidden pattern for our path of length k, we have to check its applicability to G_k. Thus, we encode the reachability check for the pattern F by $[\![F]\!]^{Cond}_k$.

Dynamics of Nodes. For illustration purposes, we omitted the aspects of added and deleted nodes. These are more difficult to handle, as they can have many side effects. We summarize the additional encoding efforts necessary. Given that we cannot handle a changing universe in our formula, we need a universe $\mathcal{U}$ that can capture all possibilities. Thus, we search the transformation rule with the highest number *max* of new nodes and add $k \cdot max$ new nodes to $\mathcal{U}$. This trick enables us to apply k transitions without the need to change the universe.

Of course this introduces the new problem of having "potential" nodes in $\mathcal{U}$ that do not actually belong to the graph. To solve this problem, we introduce the *dead*-predicate, initially valued 1 for all potential nodes and 0 for all nodes in the initial graph. The label set, the rules (including NACs) and their encodings can be automatically adjusted in a fairly straightforward manner to emulate the expected semantics of that predicate. In addition, one has to encode the deletion of all edges adjacent to deleted nodes to ensure the non-existence of dangling edges. For details, please refer to [16].

The central theorem of this paper states the correctness of our approach.

Theorem 1. *Given a GTS $\mathcal{G} = (S, P)$, a forbidden pattern F and a bound k. The encoding $[\![S]\!] \wedge [\![T]\!]_1 \wedge ... \wedge [\![T]\!]_k \wedge [\![F]\!]^{Cond}_k$ is satisfiable iff F is reachable in $\mathcal{G}$ in k steps. The satisfying interpretation represents graphs G_0 to G_k (i.e., $\mathcal{I}_i(G_i)$) in a path $G_0, ..., G_k$ of $\mathcal{G}$ containing the forbidden pattern.*

As the satisfying interpretation represents the graphs along a path with the forbidden pattern, we can directly use this to present this path.

From Reachability to LTL. The version presented above of encoding paths of length k of a graph transformation system lacks the ability of describing finite paths of length smaller k. If a path has length $i < k$, then there is no transformation rule applicable to host graph G_i. Thus, the encoding $[\![T]\!]_{i+1}$ of the transition $i + 1$ is not satisfiable, which means the path cannot be found and therefore not be checked for reachability. In consequence of this, we have to iteratively check all paths of length i with $1 \leq i \leq k$. This however forces the satisfiability checking of up to k instances, resulting in a long runtime. To overcome this issue, we introduce a self-loop, which is only applicable, if and only if no transformation rule is applicable. In order to encode this self-loop, we developed an encoding of the non-applicability of a rule. The self-loop enables a path of length smaller than k to be represented by a path of length k. As strictly smaller paths are not checked separately, we adapt the encoding of the forbidden pattern to $[\![F]\!]_0^{Cond} \vee ... \vee [\![F]\!]_k^{Cond}$. Thus, we do not need to check for strictly smaller paths. We will compare both approaches in the next section.

This self-loop and the encoding of the non-applicability allow us to completely encode the BMC of LTL formulae. The semantics and the encoding of the LTL formula are the same as for standard BMC of Kripke structures, except that atomic propositions here mean the applicability of a pattern and negated atomic propositions mean the non-applicability. In general, our technique is slightly more expressive than LTL, as we can check whether a transformation rule actually was applied in a transition, not just its potential applicability to a host graph.

Theorem 1 describes the iterative version for a forbidden pattern. All previously mentioned lemmata and a more general theorem for LTL formulae without iterative checking are described and proved by Isenberg [16].

4 Implementation and Evaluation

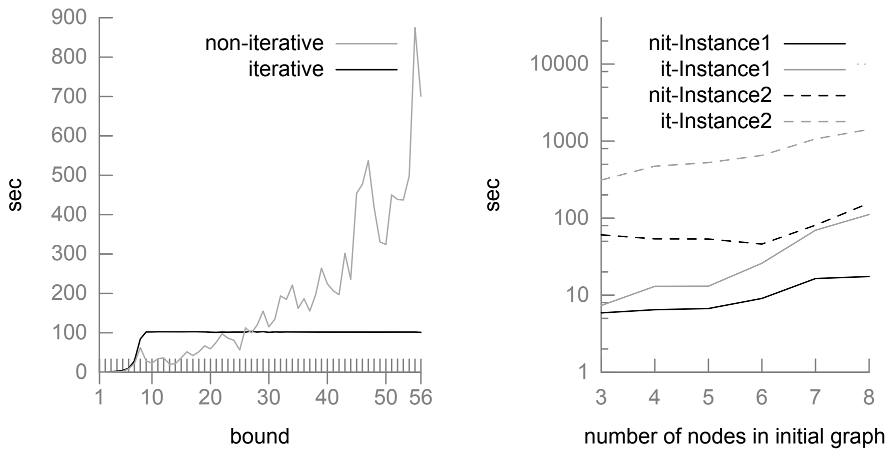
In the following, we shortly describe our implementation and present results of the evaluation of our technique.

We built a prototypical implementation of our BMC technique as an extension to GROOVE [24], which can generate the encoding in two ways, the first of which is described in this paper. The second encoding does not use different *predicates* (numbering) to differentiate between the graphs in different steps of a path, but different *universes*. A first result of the evaluation is that these encodings are basically equally well suited; none can outperform the other. In both cases the result of the encoding is an SMT formula written according to the SMT-Lib v2.0 format [5], which is then fed into the SMT solver Z3 [21]. If a satisfiable interpretation is found, the represented counterexample is displayed.

We tested our approach on several examples, including the car platooning GTS [1]. The forbidden pattern used by this example is shown in Figure 1b and does not occur within the state space of the original GTS. To check our approach, we also created a faulty specification of car platooning, essentially by omitting the NAC of rule 13 (Instance 1). With this rule modified, the graph transformation system yields several paths containing the forbidden pattern. To further increase the complexity of verification, we added a rule creating new cars as to make the state space infinite and to have an instance with node dynamics (Instance 2). These instances have 15, respectively 16 transformation rules.

On these two instances, we compare the iterative (*it*) reachability check and the non-iterative one (*nit*) using the self-loop. For this, we use several initial graphs with different numbers of nodes (see Figure 4b). In addition, we compared both the iterative and non-iterative reachability check for different bounds using Instance 1 with an initial graph consisting of 8 nodes (see Figure 4a). Note, that the shortest path containing the forbidden pattern has length 9.

Our technique works most efficiently in its non-iterative version when the guess of the bound is close to the size of the counterexample. If the bound is



(a) Instance 1 with 8 nodes in initial graph (b) Different Instances with bound 10

Fig. 4. Results of our experiments

chosen much higher than the length of the smallest counterexample, the iterative can outrun the non-iterative version, as it only evaluates paths up to the length of the smallest counterexample (see Figure 4a). One further observation gained from looking at different case studies is that the order of the encodings of the transformation rules matters (with respect to solving time), in particular when having only one path containing the forbidden pattern in a huge set of paths.

We also compared our tool with GROOVE's full state space exploration as well as its BMC technique. The BMC technique in its version as online in March 2013 cannot (strangely) deal with the car platooning GTS of Instance 2 which is infinite state. Unfortunately, we do not see the reason for this, and thus a meaningful comparison is not possible. A comparison with GROOVE's full state space exploration shows that GROOVE is faster for the finite state Instance 1, but – because of the very nature of the exploration technique – cannot deal with the infinite state Instance 2. The conclusion of our evaluation is thus that our technique can be seen as a complement to existing approaches, especially useful for bug finding in GTSs with infinite state spaces. We moreover think that our approach will prove its strength in cases when the error path is already approximately known, for instance when checking for spurious counterexamples.

5 Related Work

BMC [8] is a standard technique originally from the field of circuit verification. It unfolds the transition relation up to a predefined boundary to check for errors. One of its main concepts is the encoding of the problem as a SAT instance.

While the encoding of problems as a SAT/SMT problem is a well known standard technique in the field of planning [18,26], this is not the case for graph transformation systems. However, there are some approaches transforming GTSs into the planning language PDDL [29] to use recent progress in that field.

There is also research focusing on transformations into other target logics to verify graph properties, e.g., using rewriting logic [30,7].

However, some approaches use only the concept of bounding the state space without the encoding as a SAT problem. Kastenberg [17] constructs a Büchi automaton on-the-fly and uses a nested-DFS algorithm to check non-emptiness by iteratively searching the state space up to predefined boundaries.

Baresi [3] transforms a given GTS into Alloy, a simple structural modeling language, which is then automatically transformed into a propositional formula. The generated formula is afterwards fed into a SAT-solver. This approach, however, doesn't utilize SAT-solving directly, but uses an intermediate tool.

Another approach using propositional formulae is the one presented by Kreowski [20], which is very similar to ours, but differs in some major points. We use an encoding in first-order predicate logic, allowing us to represent the graph transformation system in a more compact and readable manner. This can be of great help with respect to planned future research, where we are interested in interpolation. In addition, the representation as SMT-formula gives us the ability to let the SMT-solver handle the matches itself. In contrast, Kreowski enumerates all possible matches in his encoding. The overhead of using SMT instead of SAT should be small for the used theory. Thus, our approach can be fast if the solver is able to restrict the search space quickly.

Other approaches to solving the problem with state space explosion and infinite growth focus on overapproximation, rather than on bounded state spaces (underapproximation). The upside to this is that positive results extend to the entire state space, rather than just a small subset that has been examined. The downside is that negative results might not be definite and there can be no guarantee of termination without excluding some types of inputs.

Into this class of approaches falls the work by Barbara König et al. [19], which uses a combined formalism of graphs and Petri nets. Other overapproximation approaches, inspired by shape analysis [27], use a more direct encoding of abstraction. Examples are the work by Rensink and Zambon [23,31] as well as Steenken, Wonisch and Wehrheim [28].

Another approach that performs a kind of overapproximation is given by Giese, Beyer et al. [6]. Here the idea is to prove inductive invariants of GTS by starting at the error pattern and working backwards.

6 Conclusion

In this paper, we presented a technique for BMC of graph transformation systems via SMT solving. To this end we encoded the reachability problem of a forbidden pattern in a GTS as an SMT formula. The presented approach can easily be extended to cover properties specified in LTL, and we have already implemented this extension as well.

One idea for further research could be to optimize the way of handling new nodes in transformation rules. In addition, one could try to find good heuristics for arranging the encodings of the transformation rules to improve the speed even further. Our main objective for the future is, however, the usage of this approach for a fast counter example analysis in our shape analysis based verification technique for GTS [28].

References

1. Backes, P., Reineke, J.: A graph transformation case study for the topology analysis of dynamic communication system. In: Transformation Tool Contest 2010. CTIT Workshop Proceedings, vol. WP10-03, pp. 107–118. University of Twente (2010)
2. Baldan, P., König, B., Rensink, A.: Summary 2: Graph grammar verification through abstraction. In: König, B., Montanari, U., Gardner, P. (eds.) Graph Transformations and Process Algebras for Modeling Distributed and Mobile Systems. Dagstuhl Seminar Proceedings, vol. 04241 (2004)
3. Baresi, L., Spoletini, P.: On the use of Alloy to analyze graph transformation systems. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 306–320. Springer, Heidelberg (2006)
4. Barrett, C., Deters, M., de Moura, L., Oliveras, A., Stump, A.: 6 Years of SMT-COMP. Journal of Automated Reasoning, 1–35 (2012), 10.1007/s10817-012-9246-5
5. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories, Edinburgh, UK (2010)
6. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic invariant verification for systems with dynamic structural adaptation. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) ICSE, pp. 72–81. ACM (2006)
7. Bergmann, G., Boronat, A., Heckel, R., Torrini, P., Ráth, I., Varró, D.: Advances in model transformations by graph transformation: Specification, execution and analysis. In: Wirsing, M., Hölzl, M. (eds.) SENSORIA. LNCS, vol. 6582, pp. 561–584. Springer, Heidelberg (2011)
8. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without bDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
9. Bisztray, D., Heckel, R., Ehrig, H.: Verification of architectural refactorings by rule extraction. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 347–361. Springer, Heidelberg (2008)
10. Bouton, T., Caminha B. de Oliveira, D., Déharbe, D., Fontaine, P.: veriT: An open, trustable and efficient smt-solver. In: Schmidt, R.A. (ed.) CADE 2009. LNCS, vol. 5663, pp. 151–156. Springer, Heidelberg (2009)
11. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An Interpolating SMT Solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012)
12. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013)
13. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars, pp. 247–312. World Scientific (1997)

14. Engels, G., Güldali, B., Soltenborn, C., Wehrheim, H.: Assuring consistency of business process models and web services using visual contracts. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 17–31. Springer, Heidelberg (2008)
15. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundam. Inform. 26(3/4), 287–313 (1996)
16. Isenberg, T.: Bounded Model Checking für Graphtransformationssysteme als SMT-Problem. Master's thesis, University of Paderborn, Germany (2012)
17. Kastenberg, H.: Graph-based software specification and verification. Ph.D. thesis, University of Twente, Enschede (October 2008)
18. Kautz, H.A., Selman, B.: Planning as satisfiability. In: ECAI, pp. 359–363 (1992)
19. König, B., Kozioura, V.: Augur 2 - a new version of a tool for the analysis of graph transformation systems. Electronic Notes in Theoretical Computer Science 211(0), 201–210 (2008); Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)
20. Kreowski, H.-J., Kuske, S., Wille, R.: Graph Transformation Units Guided by a SAT Solver. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 27–42. Springer, Heidelberg (2010)
21. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
22. Rensink, A.: The joys of graph transformation. Nieuwsbrief van de Nederlandse Vereniging voor Theoretische Informatica 9 (2005)
23. Rensink, A., Zambon, E.: Neighbourhood abstraction in GROOVE. Electronic Communications of the EASST 32 (2011)
24. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
25. Rensink, A., Zambon, E.: Pattern-based graph abstraction. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 66–80. Springer, Heidelberg (2012)
26. Rintanen, J.: Planning and sat. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 483–504. IOS Press (2009)
27. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
28. Steenken, D., Wehrheim, H., Wonisch, D.: Sound and Complete Abstract Graph Transformation. In: Simao, A., Morgan, C. (eds.) SBMF 2011. LNCS, vol. 7021, pp. 92–107. Springer, Heidelberg (2011)
29. Tichy, M., Klöpper, B.: Planning self-adaption with graph transformations. In: Schürr, A., Varró, D., Varró, G. (eds.) AGTIVE 2011. LNCS, vol. 7233, pp. 137–152. Springer, Heidelberg (2012)
30. Vandin, A., Lluch-Lafuente, A.: Towards a maude tool for model checking temporal graph properties. ECEASST 41 (2011)
31. Zambon, E., Rensink, A.: Graph subsumption in abstract state space exploration. arXiv preprint arXiv:1210.6413 (2012)

Verification of Directed Acyclic Ad Hoc Networks

Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Othmane Rezine

Uppsala University

Abstract. We study decision problems for parameterized verification of a formal model of ad hoc networks. We consider a model in which the network is composed of a set of processes connected to each other through a directed acyclic graph. Vertices of the graph represent states of individual processes. Adjacent vertices represent single-hop neighbors. The processes are finite-state machines with local and synchronized broadcast transitions. Reception of a broadcast is restricted to the immediate neighbors of the sender process. The underlying connectivity graph constrains communication pattern to only one direction. This allows to model typical communication patterns where data is propagated from a set of central nodes to the rest of the network, or alternatively collected in the other direction. For this model, we consider decidability of the control state reachability (coverability) problem, defined over two classes of architectures, namely the class of all acyclic networks (for which we show undecidability) and that of acyclic networks with a bounded depth (for which we show decidability). The decision problems are parameterized both by the size and by the topology of the underlying network.

1 Introduction

The analysis and verification of models for wireless ad hoc networks have attracted much interest in recent years [4,11,2,3,9,8,12,13,10]. Such networks usually consist of arbitrary numbers of nodes that communicate wirelessly in arbitrarily configured networks. Several features in their behaviors make them both attractive and difficult from the point of view of verification. First, the network infrastructure can be static or dynamic but is usually not a priori defined. Also, the communication between nodes occurs via broadcast over the shared radio channel medium. Messages broadcasted by a given node are only received by nodes in its proximity, in contrast to classical broadcast communication in which *all* processes of the system are able to receive the sent messages. Furthermore, since the systems may contain unbounded numbers of processes, and since the protocols are supposed to work independently from specific configurations of the network, we need to perform *parameterized verification* where we prove correctness of the system regardless of the number of nodes or the topology of the network.

Using a model similar to that proposed in [2], we view the network as a graph of nodes, where each node runs an instance of a given finite-state process.

The graph defines the underlying connectivity of the network, while a process models the code of a given fixed protocol that runs on the node. For such networks, the behavior of a node can in general be specified in terms of a sequence of states with transitions corresponding to local or to broadcast operations. Local transitions are internal to a node and do not affect the states of the other nodes in the network. Broadcast transitions on the other hand may have an impact on other nodes in the network. More precisely, we consider *selective broadcast* transitions that involve a sender (the broadcasting node), together with a set of receivers composed of the nodes that are in the topological vicinity of the sender, and that are willing to receive the broadcasted message. The vicinity of a node is defined by the underlying communication graph of the network. The broadcasting and the reception of the message happens synchronously for all involved nodes, i.e., the sender and all potential receivers in its vicinity. The interleaving semantics of our formalism does not take into account problems that could arise at the physical and link layer, such us message collision for example. We are here more interested in network and application layer protocols where these type of problems are abstracted away.

As argued in [2,3], the *control state reachability problem* or the *coverability problem* seems to be adequate for capturing several interesting properties that arise in parameterized verification of ad hoc networks. The problem consists in checking whether the system can start from a given initial configuration and evolve to reach a configuration in which at least one of the processes is in a given state. Since we are performing parameterized verification, the number of nodes that has to be handled in the analysis is not a priori bounded. In other words, we are dealing with the verification of an infinite-state system. Indeed, it is shown in [2] that the coverability problem is undecidable in the general case. Therefore, an important line of work has been done to identify classes of network topologies for which algorithmic verification is at least theoretically possible [3]. This paper proposes one such a class of topologies where the underlying graph is acyclic, and hence the communication from a node to another goes only through one direction. Such patterns arise, for instance, in the context of *Wireless Sensor Networks* (WSN), where small wireless devices are distributed over an area in order to perform different types of measurements (temperature, humidity, etc.). In WSN, it is common that the topology is static over time. Furthermore, it is also common in WSN that communication follows a specific direction; for instance this is the case for flooding protocols at the network layer [7], and for the optimized *directed diffusion protocol* [6].

From the verification point of view, we show that the coverability problem is undecidable even in the case where the network topology is acyclic (section 5). We show the undecidability result through a reduction from an undecidable problem for finite-state transducers (section 4). Then, we consider a restricted version of the problem where we assume that the depth of the acyclic graph is bounded by a given natural number k. In fact, we are still dealing with an infinite state-system since we may have an unbounded number of nodes, and an unbounded in- and out-degrees for the nodes of the graph. For this case we show decidability of the coverability problem. The proof is carried out in several steps. First we reduce the problem from the case of general acyclic graphs to that of inverted forests (forests with all edges reversed) and then to the case of inverted trees (section 6). For the case of inverted trees, we propose a novel symbolic representation of infinite sets of configurations. This symbolic representation amounts to having "higher-order multisets" in which a multiset of a certain order contains multisets of lower orders (section 7). We show that this allows to define a symbolic backward reachability analysis based on a non-trivial instantiation of the the framework of well quasi-ordered transition systems [1].

2 Preliminaries

In this section, we introduce some basic definitions and notations that we will use in the rest of the paper.

We use $\mathbb{N}$ and $\mathbb{N}_{>0}$ to denote the sets of natural numbers and positive natural numbers, respectively. Given a finite set A, we use $|A|$ to denote the number of elements in A. We use $A^{\otimes}$ to denote the set of finite multisets over A; and use A^* to denote the set of finite words over A. For words $w_1, w_2 \in A^*$ we use $w_1 \cdot w_2$ to denote the concatenation of w_1 and w_2. Sometimes, we write multisets as lists, e.g., $[a, a, b, b, b]$ is a multiset with two occurrences of a and three occurrences of b. A *quasi-ordering* (*ordering* for short) $\sqsubseteq$ on a set A is a reflexive and transitive binary relation over A (i.e. $\sqsubseteq \subseteq A \times A$, $a \sqsubseteq a$ and $a \sqsubseteq b, b \sqsubseteq c \Rightarrow a \sqsubseteq c$ for any $a, b, c \in A$). We extend the ordering $\sqsubseteq$ on A to an ordering $\sqsubseteq^{\otimes}$ on the set $A^{\otimes}$ of multisets over A such that $[a_1, \ldots, a_m] \sqsubseteq^{\otimes} [b_1, \ldots, b_n]$ if there is an injection $h : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}$ with $a_i \sqsubseteq b_{h(i)}$ for all $i : 1 \leqslant i \leqslant m$. Given a function $f : A \mapsto \mathbb{N}$, we define $\max(f) := max\{f(e) | e \in A\}$ to be the largest value taken by f over A. For a function $f : A \mapsto B$, we use $f[a \leftarrow b]$ to denote the function f' such that $f'(a) = b$ and $f'(a') = f(a')$ if $a' \neq a$.

A *(directed) graph* is a pair $G = \langle V, E \rangle$ where V is a finite set of *vertices* and $E \subseteq V \times V$ is the set of *edges*. Two graphs $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$ are said to be disjoint iff $V_1 \cap V_2 = \emptyset$. For vertices $u, v \in V$, we use $u \rightsquigarrow_G v$ to denote that $\langle u, v \rangle \in E$, use $\overset{*}{\rightsquigarrow}_G$ to denote the reflexive transitive closure of $\rightsquigarrow_G$, and use $\overset{+}{\rightsquigarrow}_G$ to denote the transitive closure of $\rightsquigarrow_G$. A *path* in G is a finite sequence $\pi = v_1 v_2 \cdots v_k$ where $v_i \rightsquigarrow_G v_{i+1}$ for all $i : 1 \leqslant i < k$. We define *first* $(\pi) := v_1$ and *last* $(\pi) := v_k$. Notice that $u \overset{*}{\rightsquigarrow}_G v$ iff there is a finite path in G with *first* $(\pi) = u$ and *last* $(\pi) = v$. For a vertex $v \in V$, we define $succ_G(v) := \{u | v \rightsquigarrow_G u\}$ to be its set of *successor* vertices, and define $pred_G(v) := \{u | u \rightsquigarrow_G v\}$ to be its set of *predecessor* vertices. For a graph $G = \langle V, E \rangle$, we define its transpose $G^{Transp} := \langle V, E^{Transp} \rangle$, where $E^{Transp} := \{\langle v, u \rangle | \langle u, v \rangle \in E\}$. In other words G^{Transp} is G with all edges reversed. We say that G is a DAG if there are no cycles in G, i.e., there are no vertices $v \in V$ with $v \overset{+}{\rightsquigarrow}_G v$. Fix a DAG $G = \langle V, E \rangle$. We define $\#G := |\{v \in V | |succ_G(v)| > 1\}|$. In other words, it is the number of vertices in the graph whose set of successors contains more than one element. For a vertex $v \in V$, we define $height_G(v) := 0$ if $succ_G(v) = \emptyset$, and define $height_G(v) := 1 + \max_{u \in succ_G(v)}(height_G(u))$

otherwise. We define $height(G) := \max_{v \in V} height_G(v)$, i.e., it is the length of a longest path in G. We define $depth_G(v) := 0$ if $pred_G(v) = \emptyset$, and define $depth_G(v) := 1 + \max_{u \in pred_G(v)}(depth_G(u))$ otherwise. We define $depth(G) := \max_{v \in V} depth_G(v)$. A *leaf* of G is a vertex $v \in V$ with height zero, i.e., $succ_G(v) = \emptyset$. We use $leaves(G)$ to denote the set of leaves of G. A *forest* is a DAG such that for all distinct pairs of vertices $v, u \in V$ we have $succ_G(v) \cap succ_G(u) = \emptyset$. A *tree* is a forest such that $|\{v|\ pred_G(v) = \emptyset\}| = 1$. We say that G is an *inverted forest/tree* if G^{Transp} is a forest/tree. The *root* of an inverted tree G is the unique vertex v with $succ_G(v) = \emptyset$. Notice that a DAG G is an inverted forest iff $\#G = 0$, i.e., it does not contain any vertices with multiple successors.

3 Directed Acyclic Ad-Hoc Networks

A Directed Acyclic Ad-Hoc Network (DAAHN) contains a finite (but arbitrary) number of nodes that are organized in a DAG. The vertices of the DAG represent individual processes, while the DAG models the topology of the network. The processes are modeled as finite-state automata that can perform both local and synchronized broadcast transitions. The successors of a vertex are the set of processes that are able to "hear" broadcast messages issued by the vertex. Depending on its local state, a successor may participate in the broadcast transition or not. Below, we describe the syntax and the operational semantics of a DAAHN, and then define two decision problems for the model related to reachability properties.

Syntax. An *Ad-Hoc Network* (AHN) consists of a pair $N = \langle P, G \rangle$ where P is a finite-state automaton describing the behavior of each process, and $G = \langle V, E \rangle$ is the communication graph between the processes. A *process* P is a tuple $\langle Q, \Sigma, \Delta, q_{init} \rangle$ where Q is a finite set of states, Σ is a finite message alphabet, $q_{init} \in Q$ is the initial state, and $\Delta \subseteq Q \times (\{\tau\} \cup \{b(m)|\ m \in \Sigma\} \cup \{r(m)|\ m \in \Sigma\}) \times Q$ is the transition relation. Intuitively, τ represents a local (internal) transition of the process. The operation $b(m)$ corresponds to broadcasting a message m, while $r(m)$ corresponds to receiving the message m. We say that N is a DAAHN if G is a DAG.

Operational Semantics. We give the operational semantics by defining the transition system induced by N. A *configuration* c of N is a function $c : V \mapsto Q$ that defines, for each vertex $v \in V$ (i.e., a process position), a state $q \in Q$. We use $q \in c$ to denote that there exists a vertex $v \in V$ such that $c(v) = q$. We use C to denote the set of configurations of N, and define the *initial configuration* c_{init} such that $c_{init}(v) = q_{init}$ for all $v \in V$. We define a transition relation $\rightarrow_N$ on the set C by $\rightarrow_N := \bigcup_{t \in \Delta} \xrightarrow{t}$, where $\xrightarrow{t}$ describes the effect of performing the transition t. Given two configurations $c, c' \in C$, we have $c \xrightarrow{t}_N c'$ if one of the following conditions holds:

– *Local Transition.* There is a $v \in V$ such that $t = \langle c(v), \tau, c'(v) \rangle \in \Delta$ and for every $v' \in V \setminus \{v\}$, we have that $c'(v') = c(v')$. A local transition modifies only the state of the involved process.

– *Broadcast.* There are $v \in V$ and $m \in \Sigma$ such that $t = \langle c(v), b(m), c'(v)\rangle \in \Delta$, and for every $v' \in V \backslash \{v\}$ one of the following conditions holds:
  - $v \rightsquigarrow_G v'$ and $\langle c(v'), r(m), c'(v')\rangle \in \Delta$.
  - $v \rightsquigarrow_G v'$, $\langle c(v'), r(m), q\rangle \notin \Delta$ for any $q \in Q$, and $c'(v') = c(v')$.
  - $v \not\rightsquigarrow_G v'$ and $c'(v') = c(v')$.

  In a broadcast transition, any successor of the sender process that can receive the message m is obliged to participate in the transition.

For both types of transitions, the topology of the system is not affected. We use $\xrightarrow{*}_N$ to denote the reflexive transitive closure of $\rightarrow_N$. A (finite) *run* ρ of N is a sequence $c_0 c_1 \ldots c_n$ of configurations such that $c_0 = c_{init}$ and $c_i \rightarrow_N c_{i+1}$ for $i : 0 \leqslant i < n$. We use $last(\rho)$ to denote c_n. A configuration c is said to be *reachable* in N if there is a run ρ of N such that $last(\rho) = c$ (notice that this is equivalent to $c_{init} \xrightarrow{*}_N c$). A state $q \in Q$ is said to be reachable in N if $q \in c$ for some reachable configuration c.

Decision Problems. The *state reachability problem* or *coverability problem* Cover is defined by a process $P = \langle Q, \Sigma, \Delta, q_{init}\rangle$ and a state $target \in Q$. The task is to check whether there is a Dag G such that $target$ is reachable in the Daahn $N = \langle P, G\rangle$. The *bounded state reachability problem* Bounded-Cover is defined by a process $P = \langle Q, \Sigma, \Delta, q_{init}\rangle$, a state $target \in Q$, and a natural number $k \in \mathbb{N}$. The task is to check whether there is a Dag G with $height(G) \leqslant k$ such that $target$ is reachable in the Daahn $N = \langle P, G\rangle$.

4 Transducers

We recall the standard definition of transducers and an undecidable problem for them. A *(finite-state) automaton* is a tuple $A = \langle Q, \Sigma, \Delta, q_{init}, Q_{final}\rangle$ where Q is a finite set of states, Σ is a finite alphabet, $q_{init} \in Q$ is the initial state, $Q_{final} \subseteq Q$ is the set of final states, and $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation. We define the language $L(A)$ of A as usual. A *(finite-state) transducer* $T = \langle Q, \Sigma, \Delta, q_{init}, Q_{final}\rangle$ is of the same form as a finite-state automaton except that $\Delta \subseteq Q \times \Sigma \times \Sigma \times Q$. Thus, a member of $L(T)$ is a word of pairs over Σ, i.e., a member of $(\Sigma^2)^*$. A transducer T induces a binary relation $R(T)$ on the set Σ^* such that $\langle a_1 \cdots a_n, b_1 \cdots b_n\rangle \in R(T)$ if $\langle a_1, b_1\rangle \cdots \langle a_n, b_n\rangle \in L(T)$. For a word $w \in \Sigma^*$, we define $T(w) := \{v | \langle w, v\rangle \in R(T)\}$. For a set W of words, we define $T(W) := \cup_{w \in W} T(w)$. Given an automaton A and a transducer T (with identical alphabets Σ) we define $T(A) := T(L(A))$. For a natural number $i \in \mathbb{N}$ and a word $w \in \Sigma^*$, we define $T^i(w)$ inductively by $T^0(w) := \{w\}$ and $T^{i+1}(w) := T(T^i(w))$. In other words, it is the result of i applications of the relation induced by T on w. We extend the definition of T^i to sets of words in the expected manner. For an automaton A, we define $T^i(A) := T^i(L(A))$.

An instance of the problem TRANSD consists of two automata A and B, and a transducer T, all with identical alphabets Σ. The task is to check whether there is an $i \in \mathbb{N}$ such that $T^i(A) \cap L(B) \neq \emptyset$. It is straightforward to show undecidability of TRANSD through a reduction from a certain non-trivial problem for Turing machines, namely whether a given Turing machine M will eventually print a given symbol a on its tape. More precisely, we use $L(A)$ to describe an appropriate encoding of (i) an empty tape of M, and (ii) the initial position of its head on the tape. The transducer T encodes one move of M, by non-deterministically guessing the position of the head, and then (i) moving the head, (ii) changing one symbol on the tape, and (iii) changing its state, according to the transition relation of M. Finally, the automaton B accepts all words that contains the symbol a.

5 Undecidability of Cover

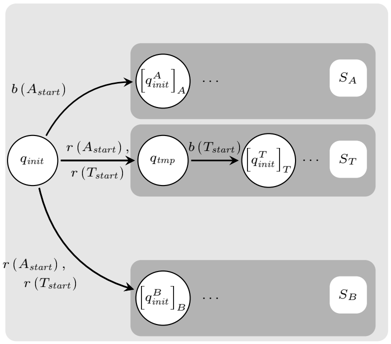
In this section, we prove the following theorem.

Theorem 1. COVER *is undecidable.*

We show undecidability through a reduction from TRANSD to COVER. Consider an instance of TRANSD defined by automata $A = \left\langle Q^A, \Sigma_A, \Delta^A, q^A_{init}, Q^A_{final} \right\rangle$ and $B = \left\langle Q^B, \Sigma_B, \Delta^B, q^B_{init}, Q^B_{final} \right\rangle$, and transducer $T = \left\langle Q^T, \Sigma_T, \Delta^T, q^T_{init}, Q^T_{final} \right\rangle$ (with $\Sigma_A = \Sigma_B = \Sigma_T$). We define a process $P = \langle Q, \Sigma, \Delta, q_{init} \rangle$ and a state $q_{accept} \in Q$ such that there is a DAG G with q_{accept} reachable in the DAAHN $N = \langle P, G \rangle$ iff there is an $i \in \mathbb{N}$ such that $T^i(A) \cap L(B) \neq \emptyset$. The manner in which we define process P (see below) will allow it to simulate both automata A and B and transducer T. The set Q of states of P is defined to be the union of four disjoint sets $Q := \{q_{init}, q_{error}, q_{accept}\} \cup S_A \cup S_B \cup S_T$ described below. The idea of the simulation is that a group of processes in N tries to build a "chain" (of some size, say i), where the root of the chain simulates A, the $(i-2)$ processes in the middle of the chain simulate T, and the last process simulates B. We will refer to such a chain as *transduction chain* below.

Simulating A. The states in S_A are used by P to simulate the automaton A. Each state $q \in Q^A$ in A has a copy $[q]_A$ in S_A. At state q_{init}, the process P may decide to simulate the automaton A (Figure 1), thus becoming the first vertex in a potential transduction chain. It does this by performing the transition $\langle q_{init}, b(A_{start}), [q^A_{init}]_A \rangle \in \Delta$ in which it moves to (the copy of) of the initial state of A. At the same time, it issues a broadcast message $b(A_{start})$ to notify its successor processes in G that it has started the simulation of A. For each transition $\langle q_1, a, q_2 \rangle \in \Delta^A$ in A there is a transition

$\langle [q_1]_A, b(a), [q_2]_A \rangle \in \Delta$ in which P simulates changing of states in A and broadcasts the symbol a to its successors. Finally, for each final state $q \in Q^A_{final}$, there is a transition $\langle [q]_A, b(m_{end}), q^A_{end} \rangle \in \Delta$ in which P declares that it has ended the simulation of A (by broadcasting the message m_{end}), after which P stops (there are no outgoing transition from q^A_{end}). Thus, in this mode, the process P broadcasts a sequence of messages corresponding to a word in $L(A)$ followed by the end-marker m_{end}.



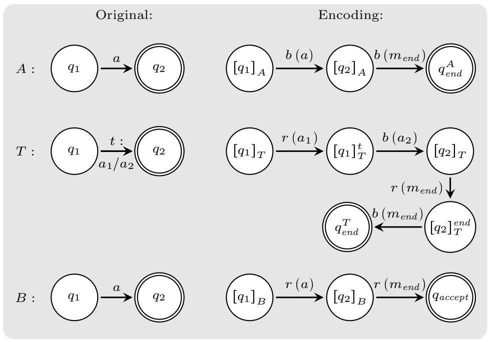


Fig. 1. Process P: initial choices **Fig. 2.** Transition and accepting state encoding

Simulating T. The states in S_T are used by P to simulate the transducer T. Each state $q \in Q^T$ in T has several corresponding states in S_T. More precisely, it has one copy $[q]_T$ (as in the case of A above); together with one temporary state $[q]^t_T$ for each transition $t = \langle q, a_1, a_2, q' \rangle \in \Delta^T$, i.e., for each transition whose source state is q. At state q_{init}, if the process P receives a message A_{start} or T_{start} from one of its predecessors, then it may decide to simulate the transducer T (Figure 1). It does so by (i) first performing one of the transitions $\langle q_{init}, r(A_{start}), q_{tmp} \rangle \in \Delta$ and $\langle q_{init}, r(T_{start}), q_{tmp} \rangle \in \Delta$, where $q_{tmp} \in S_T$ is a temporary state, followed by (ii) performing the transition $\langle q_{tmp}, b(T_{start}), [q^T_{init}]_T \rangle \in \Delta$ in which it moves to the first copy of the initial state of T. At the same time, it issues a broadcast message $b(T_{start})$ to its successors declaring that it has started the simulation of T. Intuitively, if P has received A_{start}, it will be the second process in a transduction chain (its predecessor will be the first since it simulates A), while if it has received T_{start}, it will be the $(k+1)$-th process in the chain (its predecessor will be the k-th process and the predecessor also simulates T). For each transition $t = \langle q_1, a_1, a_2, q_2 \rangle \in \Delta^T$ in T there are two transitions $\left\langle [q_1]_T, r(a_1), [q_1]^t_T \right\rangle \in \Delta$ and $\left\langle [q_1]^t_T, b(a_2), [q_2]_T \right\rangle \in \Delta$. Here, P receives the message a_1 from its predecessor (in the chain), and sends a_2 to its successors. Although, a node may have several predecessors, only one of them is allowed to act as the predecessor of the current node in the chain. This is ensured by transitions $\langle q, r(A_{start}), q_{error} \rangle \in \Delta$

and $\langle q, r(T_{start}), q_{error}\rangle \in \Delta$ for each $q \in S_T$. In other words, if the current process has already received a message from one predecessor (and thus moved to a state in S_T) then it moves to the state q_{error} if it later receives messages from any of its other predecessors. The process P then immediately suspends the simulation (there are no outgoing transition from q_{error}). Also, the process is not allowed to be "disturbed" by its predecessor while it is in the temporary state q_{tmp} or in one of the temporary states of the form $[q]_T^t$. This is due to the fact that the process in such a temporary state has not yet had time to perform the next broadcast, and therefore it is not yet ready to receive the next message form the predecessor (if this is not done, such a message will be lost in the simulation). To encode this, we add extra transitions $\langle q, r(a), q_{error}\rangle$ for each temporary state q and each message a. Also, in order to discard any sequence of received messages that do not correspond to a valid T input word, we add in Δ the transition $\langle [q]_T, r(a), q_{error}\rangle$ for every state q of Q^T and for every message $a \in \Sigma_T$ such that there is no $q' \in Q^T$ such that $\langle q, a, q'\rangle \in \Delta^T$. Finally, for each final state $q \in Q^T_{final}$, there is a transition $\left\langle [q]_T, r(m_{end}), [q]_T^{end}\right\rangle \in \Delta$; and a transition $\left\langle [q]_T^{end}, b(m_{end}), q^T_{end}\right\rangle \in \Delta$ (where $[q]_T^{end}$ is a temporary state). If the process happens to be in a final state, and it receives the end-marker from its predecessor in the chain, then it ends its simulation by notifying its successor and moving to the state q^T_{end}. Thus, in this mode, the process P receives a word w from its predecessor and sends a word in $T(w)$ to its successor.

Simulating B. The states in S_B are used by P to simulate the automaton B. Each state $q \in Q^B$ in B has a copy $[q]_B$ in S_B. At state q_{init}, if the process P receives a message A_{start} or T_{start} from one of its predecessors, then it may decide to simulate the automaton B (Figure 1). It does so by performing one of the transitions $\langle q_{init}, r(A_{start}), [q^B_{init}]_B\rangle \in \Delta$ and $\langle q_{init}, r(T_{start}), [q^B_{init}]_B\rangle \in \Delta$. In either case, it moves to the (copy of) the initial state of B. For each transition $\langle q_1, a, q_2\rangle \in \Delta^B$ in B there is a transition $\langle [q_1]_B, r(a), [q_2]_B\rangle$ in which P simulates the changing of states in B and receives the symbol a from its predecessor. In a similar manner to the case of T, we also add transitions $\langle q, r(A_{start}), q_{error}\rangle \in \Delta$ and $\langle q, r(T_{start}), q_{error}\rangle \in \Delta$ for each $q \in S_B$, and $\langle [q]_B, r(a), q_{error}\rangle \in \Delta$ for every state q of B and for every message $a \in \Sigma_B$ such that there is no $q' \in Q^B$ such that $\langle q, a, q'\rangle \in \Delta^B$. Finally, for each final state $q \in Q^B_{final}$, there is a transition $\langle [q]_B, r(m_{end}), q_{accept}\rangle \in \Delta$ in which P ends the simulation of B. Thus, in this mode, the process P receives a sequence of messages corresponding to a word in $L(B)$ followed by the end-marker m_{end}. In such a case, the process moves to the state q_{accept} which means that the given instance of Cover has a positive solution.

Correctness. We show correctness of our reduction. Suppose that the given instance of Transd has a positive answer, i.e., there is an $i \in \mathbb{N}$ and a word $w \in L(A)$ such that $T^i(w) \in L(B)$. We show that there is a Dag G such that q_{accept} is reachable in the Daahn $N = \langle G, P\rangle$, where P is defined as described above. We define $G := \langle \{v_1, v_2, \ldots, v_{i+2}\}, E\rangle$, where $v_j \rightsquigarrow_G v_k$ iff $1 \leqslant j \leqslant i+1$ and $k = j+1$. In other words, the graph forms a chain with $i+2$ nodes. The process at node 1 starts simulating A eventually broadcasting the word w followed by m_{end}. The process at node 2 starts simulating T receiving the word w symbol by symbol, and eventually broadcasting the word $T(w)$ followed by m_{end}. In general, the process at node j for $j : 2 \leqslant j \leqslant i+1$ starts simulating T receiving the word $T^{j-2}(w)$ symbol by symbol, and eventually broadcasting the word $T^{j-1}(w)$ followed by m_{end}. Finally, the process at node $i+2$ starts simulating B receiving the word $T^i(w)$ symbol by symbol, and eventually moving to the state q_{accept}.

Suppose that the given instance of Cover has a positive answer, i.e., there is a Dag G such that q_{accept} is reachable in the Daahn $N = \langle G, P\rangle$. We show that there is an $i \in \mathbb{N}$ and a word $w \in L(A)$ such that $T^i(w) \in L(B)$. We do this by extracting a transduction chain. We extract the chain vertex by vertex starting by identifying the process that simulates B, then identifying the ones that simulate T, and finally identifying the one that simulates A. Recall that q_{accept} can only be reached in a process that is simulating B. Recall also that such a process can reach q_{accept} if it receives the end-marker from a predecessor process. On the other hand, it cannot receive start messages from two different predecessors before it reaches q_{accept} since this would mean that it would move to the error state q_{error} from which it cannot reach q_{accept}. This implies that the current process has a unique predecessor. Recall that the predecessor, a sending process, must be either a process simulating A or T. If the predecessor is simulating A then we can close the chain, otherwise we have found the next transducer. In the latter case, we repeat the reasoning and find the predecessor again. Let j be the length of the chain obtained in this manner ($j \geqslant 2$ since it contains at least two vertices simulating A resp. B). Define i in the instance of Transd to be $j-2$.

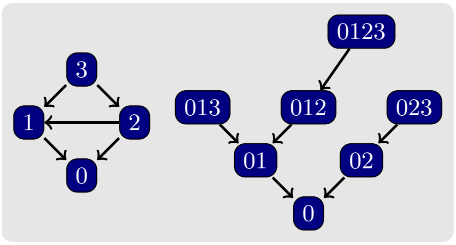
6 Forest Bounded Coverability

In this section, we show that the bounded coverability problem can be reduced from the general case of Dags to the case where we assume the Dag to be an inverted tree. We do that in two steps, namely by first reducing the problem to the case of inverted forests and then to trees.

Forests. A Daahn N is said to be an *inverted forest* if the underlying graph is an inverted forest. We consider a restricted version of Bounded-Cover, which we call Forest-Bounded-Cover. In Forest-Bounded-Cover, we require that the given Daahn is an inverted forest. We show the following theorem.

Theorem 2. Bounded-Cover *is reducible to* Forest-Bounded-Cover.

In order to prove this theorem, we first introduce a split operator over Dag*s*. Consider a Dag $G = \langle V, E\rangle$. The *split* operator splits the nodes of G, transforming it into an inverted forest. We define an inverted forest $G^\bullet := \langle V^\bullet, E^\bullet\rangle$ as follows. Each vertex $v \in V$ induces a set $v^\bullet$ in $V^\bullet$. A member of $v^\bullet$ is an inverted path π in G with



$\mathit{first}(\pi) \in \mathit{leaves}(G)$ and $\mathit{last}(\pi) = v$. The set $v^\bullet$ is defined using induction on the height of v as follows. If $\mathit{height}_G(v) = 0$ (i.e., v is a leaf) then $v^\bullet := \{v\}$. Otherwise, $v^\bullet := \{\pi \cdot v \mid \exists u. v \rightsquigarrow_G u \wedge \pi \in u^\bullet\}$. In other words, we split v into a number of copies, each corresponding to a path starting from a successor of v and ending in a leaf in G. We define $E^\bullet := \{\langle \pi_1, \pi_2 \rangle \mid \pi_1 = \pi_2 \cdot v\}$. Notice that $\mathit{height}_{G^\bullet}(u) = \mathit{height}_G(v)$ for every $v \in V$ and $u \in v^\bullet$. Therefore, $\mathit{height}(G^\bullet) = \mathit{height}(G)$. Furthermore, by definition, any vertex in $G^\bullet$ has at most one successor (no successors if it is of the form $v \in V$, or the unique successor π if is of the form $\pi \cdot v$). This means that $G^\bullet$ is an inverted forest.

Consider an instance of Bounded-Cover defined by $P = \langle Q, \Sigma, \Delta, q_{init} \rangle$, a state $\mathit{target} \in Q$, and a natural number $k \in \mathbb{N}$. We claim that the instance of Forest-Bounded-Cover defined by $P = \langle Q, \Sigma, \Delta, q_{init} \rangle$, target, and k is equivalent. For a configuration c in $N = \langle P, G \rangle$, we define $c^\bullet$ to be the configuration of $N^\bullet = \langle P, G^\bullet \rangle$ such that $c^\bullet(\pi \cdot v) = c(v)$. The following lemma shows that reachability is preserved by splitting.

Lemma 3. *If* $c_1 \rightarrow_N c_2$ *then* $c_1^\bullet \xrightarrow{*}_{N^\bullet} c_2^\bullet$.

From Lemma 3 and the fact that $c_{init}{}^\bullet(\pi \cdot v) = c_{init}(v) = q_{init}$, we conclude the following:

Lemma 4. *If c is reachable in N then $c^\bullet$ is reachable in $N^\bullet$.*

Now, we are ready to prove Theorem 2. If the given instance of Forest-Bounded-Cover has a positive answer, then the instance of Bounded-Cover has trivially a positive answer (each inverted forest is a Daahn). For the opposite direction, suppose that the instance of Bounded-Cover has a positive answer, i.e., there is a Dag G with $\mathit{height}(G) \leqslant k$ such that target is reachable in the Daahn $N = \langle P, G \rangle$. By Lemma 4, we know that target is reachable in $\langle P, G^\bullet \rangle$. The result then follows since $G^\bullet$ is an inverted forest and since $\mathit{height}(G^\bullet) = \mathit{height}(G) \leqslant k$.

Trees. We consider a yet more restricted version of Bounded-Cover, which we call Tree-Bounded-Cover. In Tree-Bounded-Cover, we require that the given Daahn is an inverted tree.

Theorem 5. Forest-Bounded-Cover *is reducible to* Tree-Bounded-Cover.

The proof of Theorem 5 is straightforward. Since the nodes inside the tree of a forest do not affect transitions of the nodes inside the other trees, we can solve Tree-Bounded-Cover for each tree separately. The given instance of Forest-Bounded-Cover has a positive answer iff Tree-Bounded-Cover has a positive answer for any of the component trees.

7 Tree Bounded Coverability

In this section, we prove the following theorem.

Theorem 6. Tree-Bounded-Cover *is decidable.*

We devote the section to the proof of Theorem 6. To do that, we instantiate the framework of *well quasi-ordered transition systems* introduced in [1]. The main ingredient of this framework is to show that the transition relation induced by the system is *monotonic* wrt. a *well quasi-ordering (wqo)* on the set of configurations. We define an ordering that we denote by $\sqsubseteq$ on configurations that are inverted trees and show monotonicity of the system behavior wrt. this ordering. Unfortunately, it is not possible to apply existing frameworks (such as the one in [1]) to directly prove the wqo of $\sqsubseteq$ on inverted trees. Therefore, we introduce a new ordering that we denote by $\sqsubseteq_2$ on a set of "higher-order multisets". We show that the ordering on higher-order multisets $\sqsubseteq_2$ is indeed a wqo and that it is equivalent to the original ordering $\sqsubseteq$ on inverted trees, which proves that $\sqsubseteq$ is itself a wqo. Then, we recall the basic concepts of the framework of well quasi-ordered systems, and show how the framework can be instantiated to prove Theorem 6.

7.1 Ordering

Assume a process $P = \langle Q, \Sigma, \Delta, q_{init} \rangle$. An *extended configuration* is a pair $e = \langle G, c \rangle$ where G is an inverted tree, and c is a configuration of the Daahn $\langle G, P \rangle$. We use $\mathbb{E}$ to denote the set of extended configurations, and, for $k \geqslant 1$, we use $\mathbb{E}^k$ to denote the set of extended configurations $\langle G, c \rangle$ where the inverted tree G is of height at most k. We define an ordering on $\mathbb{E}$ as follows. Consider extended configurations $e = \langle G, c \rangle$ with $G = \langle V, E \rangle$ and $e' = \langle G', c' \rangle$ with $G' = \langle V', E' \rangle$. For an injection $\alpha : V \mapsto V'$, we use $e \sqsubseteq^\alpha e'$ to denote that the following two conditions hold for all $v \in V$: (i) $c(v) = c'(\alpha(v))$, and (ii) If $u \rightsquigarrow_G v$ then $\alpha(u) \rightsquigarrow_{G'} \alpha(v)$. We write $e_1 \sqsubseteq e_2$ if $e \sqsubseteq^\alpha e'$ for some α. Intuitively, we can view an extended configuration as an inverted tree that is unranked (a node may have any number of predecessors) and unordered (the order in which the predecessors occur is not relevant). The ordering $e_1 \sqsubseteq e_2$ then means that the inverted tree corresponding to e_1 has a copy (an image) inside the inverted tree corresponding to e_2. In Figure 3, three extended configurations are depicted as inverted trees e_1, e_2, e_3. Here $e_1 \sqsubseteq e_2 \sqsubseteq e_3$.

7.2 Monotonicity

Given a process P, we define a transition relation $\longrightarrow$ on $\mathbb{E}$ where $\langle G, c \rangle \longrightarrow \langle G', c' \rangle$ if $G' = G$, $N = \langle P, G \rangle$, and $c \rightarrow_N c'$. The following lemma shows monotonicity of $\longrightarrow$ wrt. $\sqsubseteq$. Assume that e_1, e_2, e_3 are extended configurations.

Lemma 7. *If* $e_1 \longrightarrow e_2$ *and* $e_1 \sqsubseteq e_3$ *then there is an* e_4 *such that* $e_3 \longrightarrow e_4$ *and* $e_2 \sqsubseteq e_4$.

7.3 Higher-Order Multisets

For a finite set A and $k \geqslant 0$, we define the set $A^{\otimes k}$ inductively as follows: (i) $A^{\otimes 0} := A$; and (ii) $A^{\otimes k+1} := A^{\otimes k} \cup \left(A \times \left(A^{\otimes k}\right)^{\otimes}\right)$. In other words, a higher-order multiset of order 0 is an element in A, while a multiset of order $k+1$ is either a multiset of order k, or a pair consisting of an element in A together with a multiset of multisets of order k. Intuitively, a higher-order multiset defines an inverted tree (corresponding to an extended configuration). More precisely, a higher-order multiset of the form a represents an inverted tree consisting of a single node (labeled by a), while the higher-order multiset $\langle a, [B_1, \ldots, B_k]\rangle$ represents an inverted tree with a root labeled a, and where predecessors of the root are themselves the roots of the inverted subtrees represented by $B_1, \ldots, B_k$ respectively (see Figure 3).

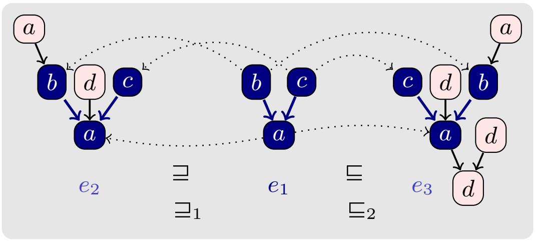


Fig. 3. The extended configurations e_1, e_2, and e_3 correspond to the higher order multisets $B_1 = \langle a, [b, c]\rangle$, $B_2 = \langle a, [\langle b, [a]\rangle, d, c]\rangle$, and $B_3 = \langle d, [\langle a, [\langle b, [a]\rangle, d, c]\rangle, d]\rangle$ respectively

We define an ordering $\sqsubseteq_2$ on $A^{\otimes k}$ in two steps. First, we define an ordering $\sqsubseteq_1$ on $A^{\otimes k}$ such that

- $a \sqsubseteq_1 a'$ if $a = a'$; and $a \sqsubseteq_1 \langle a', B\rangle$ if $a = a'$.
- $\langle a, [B_1, \ldots, B_k]\rangle \sqsubseteq_1 \langle a', [B'_1, \ldots, B'_\ell]\rangle$ if $a = a'$ and there is an injection $h : \{1, \ldots, k\} \mapsto \{1, \ldots, \ell\}$ with $B_i \sqsubseteq_1 B'_{h(i)}$ for all $i : 1 \leqslant i \leqslant k$. Notice that the second condition is equivalent to $[B_1, \ldots, B_k] \sqsubseteq_1^{\otimes} [B'_1, \ldots, B'_\ell]$.

Intuitively, $B_1 \sqsubseteq_1 B_2$ means that a copy of the inverted tree corresponding to B_1 occurs in the inverted tree corresponding to B_2 *starting* from the root. For instance, consider B_1, B_2, B_3 in Figure 3. According to the definition of $\sqsubseteq_1$, we have $B_1 \sqsubseteq_1 B_2$ while $B_1 \not\sqsubseteq_1 B_3$. This is reflected in the inverted trees corresponding to the extended configurations e_1, e_2, e_3. Although copies of e_1 occurs both in e_2 and e_3, the copy of e_1 does not start from the root of e_3. Now, we define $\sqsubseteq_2$ as follows.

- $a \sqsubseteq_2 a'$ if $a = a'$; and $a \sqsubseteq_2 \langle a', B\rangle$ if $a = a'$ or $a \sqsubseteq_2 B$.
- $\langle a, [B_1, \ldots, B_k]\rangle \sqsubseteq_2 \langle a', [B'_1, \ldots, B'_\ell]\rangle$ if one of the following two cases is satisfied:
  - $a = a'$ and there is an injection $h : \{1, \ldots, k\} \mapsto \{1, \ldots, \ell\}$ with $B_i \sqsubseteq_1 B'_{h(i)}$ for all $i : 1 \leqslant i \leqslant k$.
  - $\langle a, [B_1, \ldots, B_k]\rangle \sqsubseteq_2 B'_i$ for some $i : 1 \leqslant i \leqslant \ell$.

Notice that $\sqsubseteq_1 \subseteq \sqsubseteq_2$. Intuitively, $B_1 \sqsubseteq_2 B_2$ means that a copy of the inverted tree corresponding to e_1 occur somewhere in the inverted tree corresponding to B_2 (not necessarily starting from the root). In Figure 3, $B_1 \sqsubseteq_2 B_3$ (and a copy of e_1 occurs in e_3).

7.4 Encoding

We define an encoding function # that translates each extended configuration to a higher-order multiset. Formally, consider an extended configuration $\langle G, c\rangle$ with $G = \langle V, E\rangle$. First, we define $\#(v, c)$, for $v \in V$, by induction on $\mathit{depth}_G(v)$ as follows:

- If $\mathit{depth}_G(v) = 0$ then $\#(v, c) := c(v)$. In this case, the encoding is of order 0 (given by the state of the vertex).
- If $\mathit{depth}_G(v) > 0$ then let $\mathit{pred}_G(v) = \{v_1, \ldots, v_n\}$. Then, $\#(v, c) := \langle c(v), [\#(v_1, c), \ldots, \#(v_n, c)]\rangle$. The encoding is of the same order as the depth of the vertex; it consists of the state of the vertex itself together with the multiset of the encodings of its predecessors.

We define $\#e := \#(v, c)$ where v is the root of G. Notice that the order of $\#e$ is identical to the height of the inverted tree G. As an example, in Figure 3, if we view an inverted tree e_i, $i = 1, 2, 3$, as an extended configuration then its encoding is given by B_i. The following lemma shows that the orderings on extended configurations and higher-order multisets coincide. Let e_1 and e_2 be two extended configurations.

Lemma 8. *$e_1 \sqsubseteq e_2$ iff $\#e_1 \sqsubseteq_2 \#e_2$.*

7.5 Well Quasi-Orderings

Let A be a set and let $\sqsubseteq$ be a quasi-ordering on A. We say that $\sqsubseteq$ is *well quasi-ordering (wqo)* if it satisfies the following property: for any infinite sequence $a_0, a_1, a_2, \ldots$ of elements in A, there are $i < j$ with $a_i \sqsubseteq a_j$. We will use the following variant of Higman's Lemma [5] for our purposes:

Lemma 9. *If $\sqsubseteq$ is wqo on A then $\sqsubseteq^{\otimes}$ is a wqo on $A^{\otimes}$.*

Now, we show that the ordering $\sqsubseteq_2$ is a wqo on $A^{\otimes k}$ for any given $k \geqslant 0$. To show $\sqsubseteq_2$ is a wqo, we first show that $\sqsubseteq_1$ is a wqo on $A^{\otimes k}$ for any given $k \geqslant 0$. We use induction on k. The base case is trivial since it amounts to equality being a wqo on a finite alphabet. Consider an infinite sequence $\langle a_0, D_0\rangle, \langle a_1, D_1\rangle, \langle a_2, D_2\rangle, \ldots$ of elements in $A^{\otimes k+1}$ (notice that $D_i \in \left(A^{\otimes k}\right)^{\otimes}$). Since $a_0, a_1, a_2, \ldots$ all belong to the finite set A, there is an $a \in A$ and an infinite sequence $i_0 < i_1 < \cdots$ such that $a_{i_j} = a$ for all $j \geqslant 0$. Since $\sqsubseteq_1$ is a wqo on $A^{\otimes k}$ by the induction hypothesis, it follows by Lemma 9 that $\sqsubseteq_1^{\otimes}$ is a wqo on $\left(A^{\otimes k}\right)^{\otimes}$. By definition of wqo, there are $i_m < i_n$ with $D_{i_m} \sqsubseteq_1^{\otimes} D_{i_n}$. By definition of $\sqsubseteq_1$ we have that $\langle a_{i_m}, D_{i_m}\rangle \sqsubseteq_1 \langle a_{i_n}, D_{i_n}\rangle$.

We are now ready to show that $\sqsubseteq_2$ is a wqo. Consider an infinite sequence as the one above. Since $\langle a_{i_m}, D_{i_m}\rangle \sqsubseteq_1 \langle a_{i_n}, D_{i_n}\rangle$ and $\sqsubseteq_1 \subseteq \sqsubseteq_2$ it follows that $\langle a_{i_m}, D_{i_m}\rangle \sqsubseteq_2 \langle a_{i_n}, D_{i_n}\rangle$ which completes the proof for wqo of $\sqsubseteq_2$.

Lemma 8 implies that, for extended configurations e_1, e_2, we have that $e_1 \sqsubseteq e_2$ iff $\#e_1 \sqsubseteq_2 \#e_2$. Also, recall that, for $e = \langle G, c\rangle$ the height of G is equal to the order of $\#e$. From this and the fact that $\sqsubseteq_2$ is a wqo on $A^{\otimes k}$ for any given $k \geqslant 1$, we get the following lemma.

Lemma 10. *For any $k \geqslant 1$, the ordering $\sqsubseteq$ is a wqo on $\mathbb{E}^k$.*

7.6 Monotonic Transition Systems

A *monotonic transition system (MTS)* is a tuple $\langle \Gamma, \Gamma_{init}, \sqsubseteq, \longrightarrow, U \rangle$, where

- Γ is a (potentially infinite) set of *configurations.*
- $\Gamma_{init} \subseteq \Gamma$ is a set of *initial* configurations.
- $\sqsubseteq$ is a computable ordering on Γ, i.e., for each $\gamma_1, \gamma_2 \in \Gamma$, we can check whether $\gamma_1 \sqsubseteq \gamma_2$. Furthermore, $\sqsubseteq$ is a wqo.
- $\longrightarrow$ is a binary *transition relation* on Γ. Furthermore, $\longrightarrow$ is monotonic with respect to $\sqsubseteq$, i.e., given configurations $\gamma_1, \gamma_2, \gamma_3$ such that $\gamma_1 \longrightarrow \gamma_2$ and $\gamma_1 \sqsubseteq \gamma_3$, there is a configuration γ_4 such that $\gamma_3 \longrightarrow \gamma_4$ and $\gamma_2 \sqsubseteq \gamma_4$.
- U is defined as the upward closure $\Gamma_1 \uparrow$ of a finite set $\Gamma_1 \subseteq \Gamma$, where $\Gamma_1 \uparrow = \{\gamma' \in \Gamma |\ \exists \gamma \in \Gamma_1.\ \gamma \sqsubseteq \gamma'\}$.

We use $\xrightarrow{*}$ to denote the reflexive transitive closure of $\longrightarrow$. For sets $\Gamma_1, \Gamma_2 \subseteq \Gamma$, we say that Γ_2 is *reachable* from Γ_1 if there are $\gamma_1 \in \Gamma_1$ and $\gamma_2 \in \Gamma_2$ such that $\gamma_1 \xrightarrow{*} \gamma_2$. In the reachability problem MTS-Reach we are given an MTS $\langle \Gamma, \Gamma_{init}, \sqsubseteq, \longrightarrow, U \rangle$ and are asked the question whether U is reachable from Γ_{init}. The paper [1] gives sufficient conditions for decidability of MTS-Reach as follows. For $\Gamma_1 \subseteq \Gamma$, we define $Pre(\Gamma_1) := \{\gamma |\ \exists \gamma_1 \in \Gamma_1.\ \gamma \longrightarrow \gamma_1\}$. For $\Gamma_1 \subseteq \Gamma$, we say that $M \subseteq \Gamma_1$ is a *minor* set of Γ_1 if

- For each $\gamma_1 \in \Gamma_1$ there is $\gamma_2 \in M$ such that $\gamma_2 \sqsubseteq \gamma_1$.
- If $\gamma_1, \gamma_2 \in M$ and $\gamma_1 \sqsubseteq \gamma_2$ then $\gamma_1 = \gamma_2$.

Since $\sqsubseteq$ is a wqo, it follows that each minor set is finite. However, in general, the same set may have several minor sets. We use min to denote a function which, given $\Gamma_1 \subseteq \Gamma$, returns an arbitrary (but unique) minor set of Γ_1. We use $minpre(\gamma)$ to denote the set $\min(Pre(\{\gamma\}\uparrow))$.

It is shown in [1] that the following conditions are sufficient for decidability of MTS-Reach.

Theorem 11. MTS-Reach *is decidable if for each* $\gamma \in \Gamma$

- *we can check whether* $\gamma \in \Gamma_{init}$.
- *the set* $minpre(\gamma)$ *is finite and computable.*

7.7 From Tree-Bounded-Cover to MTS-Reach

For a natural number $k \geqslant 1$, a process $P = \langle Q, \Sigma, \Delta, q_{init} \rangle$, and a state $target \in Q$, we derive an MTS $\langle \Gamma, \Gamma_{init}, \sqsubseteq, \longrightarrow, U \rangle$ such that $\Gamma_{init} \xrightarrow{*} U$ iff there is a Dag G which is an inverted tree with $height\ (G) \leqslant k$ such that $target$ is reachable in the Daahn $N = \langle P, G \rangle$.

- Γ is the set $\mathbb{E}^k$.
- Γ_{init} is the set of pairs $\langle G, c_{init} \rangle \in \mathbb{E}^k$ and c_{init} is the initial configuration of the Daahn $\langle G, P \rangle$.
- $\sqsubseteq$ is defined on $\mathbb{E}^k$ as described above. The ordering $\sqsubseteq$ is obviously computable. Well quasi-ordering of $\sqsubseteq$ on Γ is shown in Lemma 10.

- The transition relation $\longrightarrow$ on $\mathbb{E}^k$ is defined as described above. Monotonicity is shown in Lemma 7.
- U is defined as the upward closure of the singleton $\{\langle G_1, c_1 \rangle\}$, where $G_1 = \langle \{v\}, \varnothing \rangle$ i.e., G_1 contains a single vertex v and no edges, and furthermore $c_1(v) = \mathit{target}$. Notice that U characterizes all inverted trees that contain at least one vertex labeled with *target*.

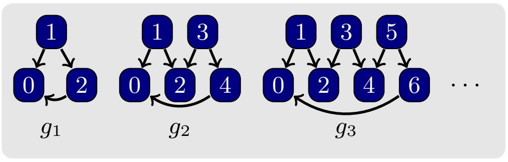
It is trivial to check whether a given configuration is initial (check whether all vertices are labeled with q_{init}). The following lemma states that the induced transition system also satisfies the second sufficient condition for decidability (see Theorem 11).

Lemma 12. *Consider the MTS defined above. Then, for each extended configuration e we can compute $minpre(e)$ as a finite set of extended configurations.*

Lemma 12, together with Theorem 11, proves Theorem 6.

8 Related Work

A fixed, generally small, number of processes has been considered when model checking techniques have been applied to verify ad hoc network protocols [4,12]. In [11] Saksena et al. define a possibly non-terminating symbolic procedure based on graph transformations to verify routing protocols for Ad Hoc Networks. Delzanno *et al.* showed in [2] that the coverability problem is undecidable in the general case of unbounded, possibly cyclic and directed graphs. In particular, the same authors considered in [3] the bounded-depth subclass of Ad Hoc Networks. Using the induced sub-graph relation on bounded-depth graphs as a symbolic representation within the well quasi-ordered transition systems framework, they proved the decidability of the coverability problem. However, this result cannot be used in the context of directed acyclic ad hoc networks because the induced sub-graph relation is not a well quasi-order in the case of the bounded depth acyclic graphs. In fact, as shown in the figure, the list of directed acyclic labeled graphs of depth 2, $g_1, g_2, g_3, \ldots$ is an infinite sequence of extended configurations of incomparable elements.



9 Conclusions

We have considered parameterized verification of ad hoc networks where the network topology is defined by an acyclic graph. We have considered the coverability problem which, for a given process definition, asks whether there is a graph and a reachable configuration where a process is in a given state. The coverability problem is used to find violations generated by a fixed set of processes independently from the global configuration. The problem turns out to be undecidable in the general case, but decidable under the restriction that the graph is of bounded depth (where the depth is bounded by a given k). Among possible directions for future work is the study of the impact of richer broadcast mechanisms such as those that allow processes to have local (unbounded) mailboxes, and to consider models augmented by timed and probabilistic transitions in order to allow quantitative reasoning about network behaviors.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.: General decidability theorems for infinite-state systems. In: LICS 1996, pp. 313–321. IEEE Computer Society (1996)
2. Delzanno, G., Sangnier, A., Zavattaro, G.: Parameterized verification of ad hoc networks. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 313–327. Springer, Heidelberg (2010)
3. Delzanno, G., Sangnier, A., Zavattaro, G.: On the power of cliques in the parameterized verification of ad hoc networks. In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 441–455. Springer, Heidelberg (2011)
4. Fehnker, A., van Hoesel, L., Mader, A.: Modelling and verification of the LMAC protocol for wireless sensor networks. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 253–272. Springer, Heidelberg (2007)
5. Higman, G.: Ordering by divisibility in abstract algebras. Proc. London Math. Soc. (3) 2(7), 326–336 (1952)
6. Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J., Silva, F.: Directed diffusion for wireless sensor networking. IEEE/ACM Trans. Netw. 11(1), 2–16 (2003)
7. Levis, P., Patel, N., Culler, D.E., Shenker, S.: Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In: NSDI, pp. 15–28 (2004)
8. Merro, M., Ballardin, F., Sibilio, E.: A timed calculus for wireless systems. Theor. Comput. Sci. 412(47), 6585–6611 (2011)
9. Nanz, S., Hankin, C.: A framework for security analysis of mobile wireless networks. Theor. Comput. Sci. 367(1-2), 203–227 (2006)
10. Prasad, K.V.S.: A calculus of broadcasting systems. Sci. Comput. Program. 25(2-3), 285–327 (1995)
11. Saksena, M., Wibling, O., Jonsson, B.: Graph Grammar Modeling and Verification of Ad Hoc Routing Protocols. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 18–32. Springer, Heidelberg (2008)
12. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: Query-based model checking of ad hoc network protocols. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 603–619. Springer, Heidelberg (2009)
13. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. Sci. Comput. Program. 75(6), 440–469 (2010)

Transducer-Based Algorithmic Verification of Retransmission Protocols over Noisy Channels

Jay Thakkar[1], Aditya Kanade[1], and Rajeev Alur[2]

[1] Indian Institute of Science
[2] University of Pennsylvania

Abstract. Unreliable communication channels are a practical reality. They add to the complexity of protocol design and verification. In this paper, we consider *noisy channels* which can corrupt messages. We present an approach to model and verify protocols which combine error detection and error control to provide reliable communication over noisy channels. We call these protocols *retransmission protocols* as they achieve reliable communication through repeated retransmissions of messages. These protocols typically use cyclic redundancy checks and sliding window protocols for error detection and control respectively. We propose models of these protocols as regular transducers operating on bit strings. Streaming string transducers provide a natural way of modeling these protocols and formalizing correctness requirements. The verification problem is posed as functional equivalence between the protocol transducer and the specification transducer. Functional equivalence checking is decidable for this class of transducers and this makes the transducer models amenable to algorithmic verification. We present case studies based on TinyOS serial communication and the HDLC retransmission protocol.

1 Introduction

Communication protocols play a foundational role in the design of distributed systems. Model checking approaches (e.g. [25]) analyze protocols for concurrency bugs. The traditional approach here is to build a finite-state model which abstracts message contents. Whether the communication channels between distributed components of a protocol deliver messages correctly or not is modeled as a non-deterministic choice. In practice, the physical channels can give rise to multiple types of faults, including message corruption, loss, and reordering. The real-world protocols, that provide reliable communication over unreliable channels, examine message contents to determine validity and semantics of messages [35]. An approach where message contents are abstracted by symbolic constants may capture semantics of such protocols only partially.

A common approach to ensure reliable communication is to combine error detection and error control mechanisms. The sender adds redundancy (checksum bits) to the messages which is used by the receiver to detect errors in the received messages. Upon receiving a corrupted message, the receiver requests the sender for retransmission of the message. We call protocols which follow this scheme as *retransmission protocols* for noisy channels. These protocols typically use cyclic redundancy checks (CRC) [29] and sliding window protocols [14, 34] for error detection and control respectively. By a *noisy channel*, we mean a channel that can corrupt messages but not drop, re-order, or duplicate them. TinyOS serial communication [1], high-level data link control (HDLC) [26], and transmission control protocol (TCP) [2] are examples of widely used retransmission protocols over noisy channels.

The objective of this paper is to devise a technique to model and verify such protocols. We observe that, for accurate modeling of the protocols, we have to encode messages as sequences of bits that are exchanged over channels. (In Section 2, we show that a protocol model, in which messages are abstracted as symbolic constants, can deliver a wrong sequence of messages.) However, verification of finite-state machines communicating asynchronously over unbounded FIFO channels is known to be undecidable [12].

In this paper, we present an algorithmic technique for verification of retransmission protocols where messages, checksums, and acknowledgements are treated as *bit strings*. In our models, the message strings and the number of (re)transmission rounds can be unbounded. Our approach is based on the observation that the protocol components can be viewed as *transductions over bit strings*. As an example, consider a sender which gets a message M from its client. The sender transmits it and in return, gets an acknowledgement a from the receiver. The combined input to the sender can be modeled as a string $M\sharp a$ where $\sharp$ is the end-marker attached to the message. The semantics of the sender is that if a is 0 (a negative acknowledgement) then it should retransmit the message. This can be formalized as a transduction f defined as $f(M\sharp 1) = M\sharp$ and $f(M\sharp 0) = M\sharp M\sharp$. We give such transducer-based semantics to protocol components (sender and receiver) and obtain the protocol model by *sequential composition* of its components. Our modeling approach thus differs from the modeling of protocol components as asynchronously communicating finite-state machines [12]. The correctness requirement here is that, in spite of message corruption, the receiver delivers only correct messages to its client and in the same order as the sender's client intended. We show that the specification can also be modeled as a transducer. The verification problem is then reduced to checking the functional equivalence of the protocol and the specification transducers.

In many cases, the transductions defined by the sender, the receiver, and the specification of retransmission protocols are *regular*. Regular transducers are closed under sequential composition [16] and equivalence checking is decidable for them [22]. This makes these transducer models amenable to algorithmic verification. Even though sequential transducers also enjoy similar properties, they are not expressive enough to model components of retransmission protocols. For example, they cannot model the sender transduction f.

In this work, we use *deterministic streaming string transducers* (SSTs) [5, 6] as finite-state descriptions of regular transductions. An SST is equipped with a finite set of string variables to store strings over the output alphabet. The output of an SST can be defined in terms of the string variables. As compared to equally expressive models of two-way transducers and MSO-definable transductions, SSTs provide a more natural way of modeling retransmission protocols. In particular, the buffers used by the sliding window protocols for storing messages for retransmission can be modeled directly as string variables of SSTs.

We have designed transducer models of TinyOS serial communication protocol and HDLC with different CRC polynomials and have implemented a prototype tool for their verification. The sliding window protocols are fairly complex and are challenging even for manual proofs [24, 23]. Our approach proposes transducers for modeling only those aspects of these protocols that are relevant for reliable communication over noisy channels and gives an algorithmic verification technique. Some features of these protocols, such as timers and dynamic window sizes, are neither relevant nor amenable to our approach.

In Section 2, we motivate transducer models of retransmission protocols with an example. The modeling approach is presented in Section 3 and the verification algorithm is discussed in Section 4. Section 5 describes case studies. The related work is surveyed in Section 6. In Section 7, we sketch future work and conclude.

2 Motivating Example

Suppose we want to transmit two messages (bit strings) M_0 and M_1 over a noisy channel. Following the usual procedure of abstraction, let us denote them respectively by symbolic constants m_0 and m_1. In this example, we use the stop-and-wait protocol for error control. In this protocol, the sender prepends one bit sequence numbers to messages. This defines the space of valid encodings as $\{0, 1\} \times \{m_0, m_1\}$. Let ack_0 and ack_1 denote acknowledgements indicating that the receiver expects a message with sequence number 0 or 1 next.

Consider a simple model of a noisy channel depicted in Fig. 1(a)–1(b). The sender-to-receiver channel is called the *forward channel* and the opposite channel is called the *backward channel*. The forward channel may corrupt the sequence bit of the encoded message as shown in Fig. 1(a). The messages to the left of the arrows are the original messages and those on the right indicate their possible incarnations at the other end of the channel. A dashed arrow indicates message corruption. The backward channel may corrupt ack_0 to ack_1 and vice versa.

The noisy channel can give rise to a sequence of message corruptions inducing the receiver to deliver an incorrect sequence of messages to its client. Fig. 1(c) shows such an example. The strings with Gray background indicate contents of the sliding window buffers. The dashed arrows annotating message exchanges indicate corruptions. In the first message transfer, the receiver cannot detect that $(1, m_0)$ is corrupt since $(1, m_0)$ does belong to the space of valid messages. It nevertheless discards it, as it is awaiting a message with sequence number 0. As ack_1 belongs the space of valid acknowledgements, the sender too cannot detect corruption and transmits the next message. Even after corruption, $(0, m_1)$ is accepted by the receiver since it has the expected sequence number. Overall, the sender believes that it has sent $\langle m_0, m_1\rangle$ but the receiver accepts $\langle m_1, m_1\rangle$.

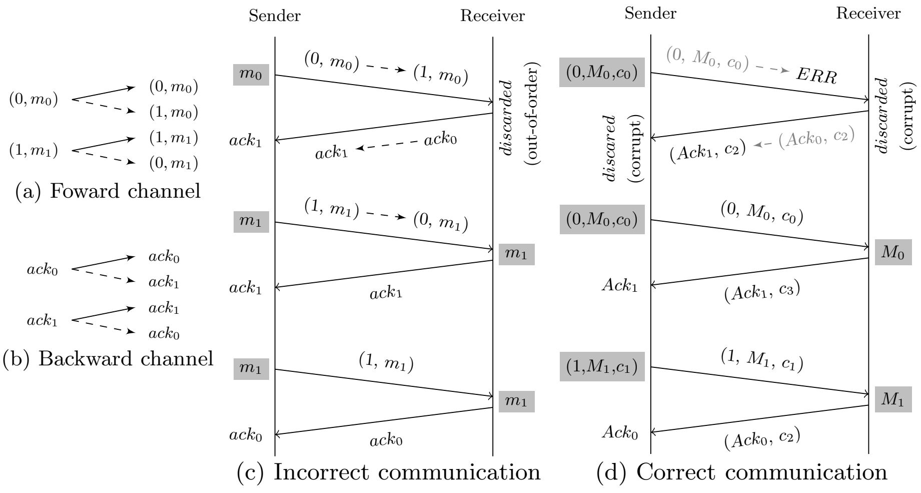


Fig. 1. A noisy channel model and scenarios of correct and incorrect communication

The problem with this protocol model is that it is unable to detect corruption. Our solution is to make the modeling more precise by treating messages, acknowledgements, and checksums as bit strings. We represent a *message encoding* as a triple (n, M, c) where n is the (fixed-length) bit encoding of a sequence number, M is a message string (of an arbitrary length), and c is the (fixed-length) checksum over strings n and M. It is possible to build finite-state protocol models by bounding the length of message strings. This could be useful for finding bugs quickly. We are however interested in verification of retransmission protocols without bounding the message length artificially.

We propose a transducer-based model that does not exhibit the incorrect communication scenario discussed above. We view the input to the sender as a sequence of messages and acknowledgements (protected by checksum). Consider the following string of inputs to the sender according to our scenario:

$$M_0\ (Ack_1, c_2)\ (Ack_1, c_3)\ M_1\ (Ack_0, c_2)$$

The acknowledgement strings corresponding to ack_0 and ack_1 are denoted by Ack_0 and Ack_1 respectively. Let the checksum be an even parity bit with c_2 and c_3 as the correct checksums for strings Ack_0 and Ack_1 respectively.

Consider a sender transducer which scans the input string in a single left-to-right pass. Here, it reads the message string M_0, generates an encoding $(0, M_0, c_0)$, and stores it in a string variable corresponding to the sliding window buffer. It then looks ahead at the acknowledgement to determine its output in the first round of transmission. It detects that (Ack_1, c_2) is an invalid/corrupt acknowledgement string. It does not know whether the message was received correctly at the receiver or not. It conservatively assumes that it was not delivered correctly. The sender sets its output to some string, say *ERR*, with an incorrect checksum – modeling the effect of corruption to M_0. Since (Ack_1, c_3) is a valid encoding, the sender sets its output in the second round to $(0, M_0, c_0)$. Finally, it reads and encodes M_1. The encoding of M_1 is appended to the output since (Ack_0, c_2) indicates its correct delivery. The output string of the sender is thus:

$$ERR\ (0, M_0, c_0)\ (1, M_1, c_1)$$

We depict the message exchanges according the input/output of the sender transducer in Fig. 1(d). It differs from the corresponding scenario in Fig. 1(c) in the output of the sender in the second round. The sender outputs (an encoding of) message M_0 instead of M_1. Thus, this sender cannot be tricked into making an incorrect transition on receiving a corrupt acknowledgement. The receiver too can be modeled as a transducer. It takes the message triples, verifies the checksum, and accepts a message if the checksum agrees. In this example, it accepts the correct message strings M_0 and M_1 and in the same order as that used by the sender, in spite of corruptions of messages and acknowledgements.

3 Transducer Models of Retransmission Protocols

We use streaming string transducers for protocol modeling and start with a brief introduction to them before giving the transducer constructions.

3.1 Streaming String Transducers

A *deterministic streaming string transducer* (SST) [5] makes a *single* left-to-right pass over an input string to produce an output string. It uses a finite set of string variables to store strings over the output alphabet. Upon reading an input symbol, it may move to a new state and update all the string variables using a parallel (simultaneous) *copyless* assignment where the right-hand side expressions are concatenations of string variables and output symbols. A parallel assignment is copyless if no string variable appears more than once in any of the right-hand side expressions. For example, let a be an output symbol and $X = \{x, y\}$ be the set of variables. Then, $[x = x.y, y = a]$ is a copyless assignment, whereas, $[x = x.y, y = y]$ is not because y occurs twice on the right-hand side.

Formally, an SST is an 8-tuple $(Q, \Sigma_1, \Sigma_2, X, F, \delta, \gamma, q_0)$ machine, where Q is a finite set of states, Σ_1 and Σ_2 are finite sets of input and output symbols respectively, X is a finite set of string variables, F is a partial output function from Q to $(\Sigma_2 \cup X)^*$ with the constraint of copyless assignment, δ is a state transition function from $(Q \times \Sigma_1)$ to Q, γ is a variable update function from $(Q \times \Sigma_1 \times X)$ to $(\Sigma_2 \cup X)^*$ using copyless assignments and $q_0 \in Q$ is the initial state. The semantics of an SST is defined in terms of the summaries of a computation of the SST. Summaries are of the form (q, s) where q is a state and s is a valuation from X to Σ_2^* which represents the effect of a sequence of copyless assignments to the string variables. The second component can be extended to a valuation from $(\Sigma_2 \cup X)^*$ to Σ_2^*. In the initial configuration (q_0, s_0), s_0 maps each variable to the empty string. The transition function is defined by

$\psi((q, s), a) = (\delta(q, a), s')$ where for each variable $x \in X$, $s'(x) = s(\gamma(q, a, x))$. For an input string $w \in \Sigma_1^*$, if $\psi^*((q_0, s_0), w) = (q, s)$, then if $F(q)$ is defined, the output string is $s(F(q))$ otherwise it is undefined.

Example. Let us consider the sender transduction f described in Section 1. This can be implemented by an SST using four states, $Q = \{p_0, p_1, p_2, p_3\}$ where p_0 is the initial state, as shown in Fig. 2. Here, $\Sigma_1 = \Sigma_2 = \{0, 1, \sharp\}$. A message, M is a string over $\{0, 1\}$ and $\sharp$ is the message end marker. The SST for this example requires two string variables, $X = \{x_1, x_2\}$, both of which store a copy of the input message M. On reading 0 (resp. 1) in p_0, the SST remains in state p_0 and appends 0 (resp. 1) to both x_1 and x_2. The state transitions and variable updates are shown in Fig. 2. On reading the end-marker $\sharp$ in state p_0, the SST moves to state p_1 and appends $\sharp$ to both x_1 and x_2. In state p_1, there can be two input symbols, 0 (a negative acknowledgement) and 1 (a positive acknowledgement). Upon seeing 1, the SST moves from p_1 to p_2 and frees x_2. The output function in state p_2 is defined to be x_1, i.e., $F(p_2) = x_1$. The contents of x_1 here is $M\sharp$. On reading 0, it goes from state p_1 to state p_3 whose output is $x_1.x_2$, i.e., $F(p_3) = x_1.x_2$. This holds the output string $M\sharp M\sharp$. The output function F is undefined for other states.

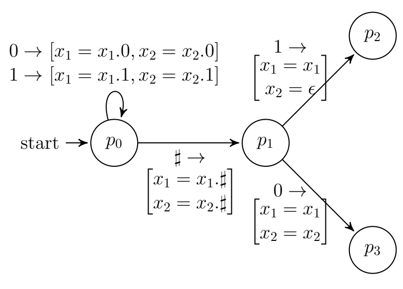

$0 \to [x_1 = x_1.0, x_2 = x_2.0]$
$1 \to [x_1 = x_1.1, x_2 = x_2.1]$
start
p_0
$\sharp \to \begin{bmatrix} x_1 = x_1.\sharp \\ x_2 = x_2.\sharp \end{bmatrix}$
p_1
$1 \to \begin{bmatrix} x_1 = x_1 \\ x_2 = \epsilon \end{bmatrix}$
p_2
$0 \to \begin{bmatrix} x_1 = x_1 \\ x_2 = x_2 \end{bmatrix}$
p_3

Fig. 2. An SST for the sender transduction f described in Section 1

3.2 Construction of Sender and Receiver Transducers

We present the transducer constructions abstractly without fixing the window sizes and CRC polynomials. An SST for a specific protocol configuration can be obtained by fixing values of these parameters. Due to the space constraints, for the sliding window logic, we consider only the *go-back-n* protocol. It is easy to extend the construction to stop-and-wait and selective-repeat protocols.

Sender SST. Let W be the size of the sliding window of the sender. The sender can store up to W outstanding messages in a set of buffers. As more than one message can be in transit, to distinguish between them, the sender associates a sequence number with each message. Let the set of sequence numbers that the sender can use be $0, \ldots, N-1$. As a noisy channel may corrupt a message, the sender attaches a checksum with each of its outstanding messages.

In our setting, a sender receives a sequence of strings over $\{msg, ack_i, b\_ack\}$ where msg is a bit string that is provided to the sender by its client for transmission, ack_i acknowledges the outstanding messages up to sequence number $i-1$, and b_ack is an acknowledgement string with incorrect checksum – modeling corruption in the backward channel. Each acknowledgement corresponds to a *(re)transmission round.* If the receiver sees a corrupt message, it drops it and resends ack_i where i is one plus the sequence number of the last message received successfully. We call such repeated acknowledgements as *retransmission* *requests*. The corruptions in the forward channel are thus identified by presence of retransmission requests in the sender's input string. The sender treats a *b_ack* as a retransmission request for all outstanding messages.

The behavior of a noisy channel, that is, message corruption, is modeled in the output function of the sender. The output of the sender is thus the sequence of messages that the receiver sees at its end of the noisy channel. It is a sequence of strings over $\{ERR, Emsg\}$ where ERR is a string with incorrect checksum, modeling a corrupt message. A string $Emsg$ is an *encoded message* consisting of concatenations of a sequence number n (represented by a bit string), the message msg being transmitted, and the checksum crc. We require some meta-symbols to separate the substrings of $Emsg$ and to separate consecutive messages and acknowledgements. For brevity, we omit them in this discussion.

For an outstanding message msg with sequence number i, the sender must decide whether the receiver sees ERR or its valid encoding $Emsg$. The encoding $Emsg$ is emitted only if the subsequent acknowledgement is ack_j where j is the sequence number that follows i. Otherwise, ERR is emitted. Thus, the sender must look ahead at the suffix of the string to determine its output. Since there are only a finite number of acknowledgement strings, they form a regular language. Thus, the sender can determine its output with a *regular look-ahead*. The output function concatenates the encoded messages and ERR strings. An encoded message is present in the output at most once, only when it is positively acknowledged. For every retransmission request for a message, the fixed string ERR is added to the output instead of the encoded message. Thus, the size of the output string is a constant multiple of the size of the input string. This ensures that the resulting transducer is regular.

Presently, we model the behavior of a noisy (forward) channel in the output of the sender. The channel can also be modeled independently as a deterministic SST. The sender can output an additional bit with a message to indicate whether it is to be delivered uncorrupted or not, based on the regular look-ahead at the acknowledgements. The channel SST could simply inspect this bit to determine its output.

Sliding Window Management. We model each buffer of the sender by a string variable. Let the set X of string variables of the sender SST be $\{x_0, \ldots, x_{W-1}\}$. We remember the sliding window configuration and the next unused sequence number S in the states of the SST. A *sliding window configuration* is a pair of numbers F and L representing the first and the last occupied buffers respectively. Fig. 3 shows a snapshot of the abstract model of the sender SST. As a convention, if a variable is unmodified, we do not show an assignment to it.

Given F, L, and S, it is straightforward to identify sequence numbers of the outstanding messages. If the sender receives ack_i then the buffers containing the messages with sequence numbers up to $i - 1$ are freed (by assigning ϵ to them). The pointer F is updated to reflect this as indicated in the transition t_3 in Fig. 3, where OM is the set of sequence numbers of the outstanding messages and R is the number of buffers that are to be freed. If ack_S arrives then all outstanding messages are delivered. The sliding window becomes empty $(F = 0, L = -1)$.

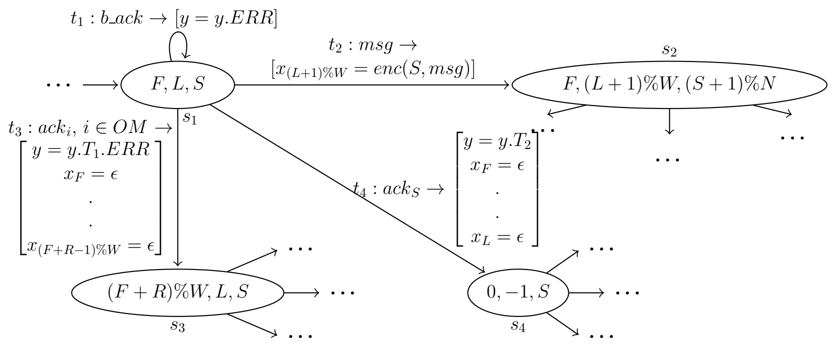


Fig. 3. A snapshot of the abstract model of the sender transducer where $T_1 = x_F.x_{(F+1)\%W} \dots x_{(F+R-1)\%W}$ and $T_2 = x_F.x_{(F+1)\%W} \dots x_L$

The transition t_4 shows this behavior. If the sliding window is not full, then the SST may read a message *msg*. It computes encoding of *msg* augmented with the sequence number S and stores the encoded message in the next free buffer. The transition t_2 illustrates this. The function *enc* denotes a CRC computation. We discuss it later in this section. A bad acknowledgement *b_ack* does not affect the state and variables of the SST as indicated by the transition t_1.

Handling Retransmission Requests. The protocol runs in potentially unbounded number of (re)transmission rounds. We wish to define the output of the sender across all the rounds as a string. We accumulate the output in a string variable y. The updates to y on every type of acknowledgement are shown in Fig. 3. If ack_S is received, all the outstanding messages are appended to y in the order of their sequence numbers. The string T_2 used in transition t_4 is defined in the caption of the figure. If ack_i is received then the messages that are acknowledged positively are appended to y. For the messages that were transmitted but not received correctly, we append ERR to y, as shown in both t_1 and t_3. The output of the sender SST is set to y for every state in which message/acknowledgement is read completely. In each of these transitions, a variable x_k is used at most once on the right-hand side of variable update γ. It is either unmodified (not shown in the figure), or assigned to y and at the same time, is reset to ϵ. Thus, the resulting assignments conform to the copyless restriction of SSTs. It is easy to see that the sender SST is deterministic.

CRC Computation. We now explain the modeling of the CRC computation, denoted by *enc* function in transition t_2. A CRC computation is parameterized with a *generator polynomial* p. If the degree of the polynomial is r, then it appends an r-bit checksum to the input string. We model all possible values of an r-bit checksum into states of the SST with the initial state corresponding to checksum of zero. We require only a single string variable x to store a copy of the input string. For a state q, representing a checksum value c, we define the output to be $x.\#.c$ where # is a separator. We construct the transitions of the SST in such a way that if the SST ends up in a state q after reading the input string w, then the checksum value c, represented by q, is the checksum of w under polynomial p. The logic for constructing the state transitions follows from the semantics of linear feedback shift register (LFSR) circuits used for implementing CRC computations [20]. For want of space, we omit the details here.

The sender needs to *verify* checksum of the acknowledgement strings to classify them into ack_i or b_ack. The construction of SST for CRC verification is similar to that of CRC computation except for the variable update γ. From an input string, only the bits corresponding to the message content are copied.

Receiver SST. The output from the sender (contents of variable y) forms the input to the receiver. The input to the receiver is thus a sequence of strings over $\{ERR, Emsg\}$. The output of the receiver is a sequence of strings over $\{msg\}$. In the go-back-n logic, the receiver accepts messages only in order.

We observe that the output of the sender contains the valid encoding of a message exactly once in its output. That is, there is no message duplication. This is because whenever a variable x_k is appended to y, it is also freed. Second, the messages are always transmitted in the same order as the order in which they are obtained from the client. Thus, there is no message reordering. These two observations simplify the design of the receiver SST. In particular, the receiver SST must only be able to distinguish between corrupt and correct messages.

The receiver SST therefore consists of only two string variables: a variable x to store the current message contents and a variable y to accumulate the output across all (re)transmission rounds. Upon receiving a message encoding, the message content msg is extracted and stored in x. If the CRC verifies then x is appended to y and freed. The output for any state of the SST is the variable y. Clearly, the receiver SST is deterministic.

3.3 Sequential Composition of Sender and Receiver Transducers

The sender and receiver SSTs represent distributed components of the protocol such that the output of the sender is the input to the receiver. These SSTs model the checksum computations, message and acknowledgement encodings, and the sliding window logic of a retransmission protocol. These are however internal details of the protocol. Our goal is to analyze the end-to-end input/output relation implemented by the protocol where the input is a sequence of strings over $\{msg, ack_i, b\_ack\}$ (same as the sender) and the output is a sequence of strings over $\{msg\}$ (same as the receiver).

SSTs are known to be closed under sequential composition [5]. Thus, the input/output relation implemented by the protocol can be represented as a *protocol* SST. Further, the protocol SST can be obtained algorithmically by the sequential composition of the sender and the receiver SSTs. Consider an SST S_2 to be composed with an SST S_1. The number of string variables in the composed SST is $2.|X_2|.|Q_2|.|X_1|$, where Q_2 is the set of states in S_2 and X_1, X_2 are the sets of string variables in S_1, S_2 respectively. It is beyond the scope of this paper to discuss the algorithm for sequential composition. We refer the reader to [5]. The sender and receiver SSTs are much simpler to define than the protocol SSTs. Thus, the approach of modeling them individually and then composing is easier than constructing the SST for a protocol directly.

4 Verification of Transducer Models

In our models, each acknowledgement b_ack or ack_i corresponds to a separate (re)transmission round. Since the input to the sender can contain an unbounded number of acknowledgement strings, the number of (re)transmission rounds encoded in our models is unbounded. The number of messages received by the sender from its client too is not bounded. Similarly, there is no bound on the length of an individual message msg. Even with these sources of unboundedness, the verification problem for our models is decidable. In this section, we present the specification mechanism and the verification approach.

Specification SST. The key property of a retransmission protocol is that the messages acknowledged by the receiver (across all rounds) are delivered to the receiver's client correctly and in the same order in which the client of the sender handed them to the sender. This property can be specified as an SST. Similar to the protocol SST, the input to the specification SST is a sequence of strings over $\{msg, ack_i, b\_ack\}$ and the output is a sequence of strings over $\{msg\}$. The *specification* SST does not encode sequence numbers and checksums. It also does not corrupt or retransmit messages. It mainly encodes the sliding window logic to interpret the acknowledgement strings and determine which strings are supposed to be delivered by the receiver to its client.

The specification SST has a similar set of states, transitions, string variables, and output function as the sender transducer. The main difference between the sender and the specification SSTs is in the variable update γ. We refer to Fig. 3 to describe the specification SST for the go-back-n protocol. The specification transducer stores a message msg as it is in a string variable along transition t_2. It neither attaches a sequence number nor a checksum with it. Since the output of the specification SST is the output of the receiver (and not that of the sender), for transitions t_1 and t_3, it does not append a corrupt message to the output string variable y. The transition t_4 remains unchanged.

Verification Approach. The verification problem is to check equivalence between the protocol and the specification SSTs. Both these transducers are deterministic. Thus, for every input string w, we want to check whether the output of the protocol and the specification SSTs are same. The equivalence checking problem for (deterministic) SSTs is decidable [6]. The input to the verification algorithm consists of the sender and receiver SSTs and the specification SST as shown in Fig. 4. As discussed in Section 3.3, the protocol SST is obtained by sequential composition of the sender and the receiver SSTs.

Equivalence Checking. We briefly outline the steps involved in equivalence checking of two SSTs. To check whether two SSTs, say S_1 and S_2, are equivalent, the equivalence checking algorithm generates a 1-counter automaton, M. The objective is for M to determine whether there is an input string w and a position p such that the output symbols of S_1 and S_2 on w differ at position p. This can be generalized to infer whether (1) there is an input string w such that the output is defined for only one of S_1 or S_2, or (2) the outputs are defined but the lengths of the output strings differ.

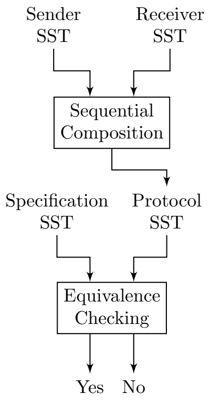


Fig. 4. Verification approach

The automaton M non-deterministically simulates S_1 and S_2 in parallel. For each of them, it guesses the position p and uses its counter to check whether the guess matches between S_1 and S_2. The complete details about the configurations of the states and transitions between them is available in [6]. A finite set, say F, of states of the automaton are identified such that if any state in F is 0-reachable in M, then the two transducers are not equivalent. Thus, the equivalence checking problem is reduced to checking 0-reachability in a 1-counter automaton. This problem is in NLOGSPACE. The number of states in M is linear in the number of states of S_1 and S_2, and exponential in the number of string variables of S_1 and S_2. Therefore, the SST equivalence problem is in PSPACE.

Decidable Extensions. Instead of emitting the error string *ERR* for corrupt messages, the sender may resend the messages themselves. In such a case, the receiver needs to eliminate duplicates. However, for the sender to be a regular transducer, the length of the output strings must be a constant times the length of the input strings. This requirement can be satisfied by considering only a *bounded number of retransmissions* and using different string variables for different rounds. Several protocols like Philips Bounded Retransmission Protocol (BRP) fall in this class of protocols [21].

Another extension can be to model the effect of a noisy channel with non-determinism using the non-deterministic SSTs (NSSTs). NSSTs are closed under sequential composition, but the equivalence problem for them is undecidable [7]. However, there is a subclass of NSSTs, called *functional* NSSTs, whose equivalence checking problem is decidable. In the future, we plan to explore of non-deterministic and bounded versions of retransmission protocols.

5 Case Studies

We model two practical protocols as case studies:

1. **TinyOS:** TinyOS is an open source real-time operating system for wireless sensor networks. Serial communication is used for host-to-mote data transfer. The SerialP [1] software module of TinyOS computes the checksum and uses the stop-and-wait protocol in the host-to-mote direction.
2. **HDLC:** HDLC [26] is a bit-oriented protocol, that operates at data link layer. The software implementation computes checksum and uses go-back-n.

Table 1. Case studies

			Sender		Receiver		Protocol		Specification	
Protocol	CRC polynomial	W	$\|Q\|$	$\|X\|$	$\|Q\|$	$\|X\|$	$\|Q\|$	$\|X\|$	$\|Q\|$	$\|X\|$
TinyOS	$z+1$	1	12	2	5	2	38	11	10	2
TinyOS	z^2+1	1	20	2	9	2	80	15	14	2
HDLC	$z+1$	2	83	3	7	2	153	24	72	3

Table 1 summarizes the three protocol configurations that we model. For each protocol, we indicate the window size (W) and the CRC polynomials used. Real-world implementations of these protocols may use polynomials of higher degree in the CRC computation. The table also shows the number of states ($|Q|$) and variables ($|X|$) required in our case studies for sender, receiver, specification and protocol SSTs. The protocol SSTs are derived by our implementation of the sequential composition algorithm.

Modeling. As an example, we present the sender and the receiver SSTs for the first protocol configuration in Table 1 (see Fig. 5). We build these according to the constructions described in Section 3. In these SSTs, if a variable update for any string variable v is not mentioned, it means that v is not updated, that is $v = v$. We use certain meta-symbols as separators: \$ to separate consecutive messages and acknowledgements, # to separate the message content from the checksum and + to indicate the start of a message. Here, ack_0 is encoded as 00, where the first 0 indicates the acknowledgement number and the second 0 indicates the checksum bit. Similarly, ack_1 is encoded as 11, whereas $b\_ack \in \{01, 10\}$ is a bad acknowledgement. The sender uses two string variables: x to store an encoding of the input message, and y to hold the output string. States q_0 to q_3 store the encoding of the input message in x, that is, $[x = enc(0, msg)]$. States q_1 and q_2 track the checksum value. State q_0 represents the empty sliding window, whereas q_3 represents the full sliding window. In state q_3, on receiving either ack_0 or b_ack, the SST appends a fixed error message, ERR (string 0.#.1.\$), to y. On receiving ack_1, the SST appends x to y, frees x and moves to the initial state for sequence number 1. The output function maps states q_0 and q_3 to y, and is undefined for other states. The part of the SST modeling sequence number 1 is similar (omitted from Fig. 5(a)).

The receiver SST shown in Fig. 5(b) needs two string variables: x to store the current message contents, and y to store the output (sequence of correctly received messages). Starting in r_0, the receiver SST first validates the sequence number but does not copy it. Then, the message content is extracted and stored in x, in states r_1 and r_2. State r_3 represents the checksum value 0 and state r_4 represents the checksum value 1. Thus r_3 indicates that the received input message encoding is not corrupt. So, after receiving the end symbol \$ in r_3, x is appended to y, and x is freed. State r_4 says that the received message is corrupt, and leaves y unchanged on receiving \$. The output function maps state r_0 to y,

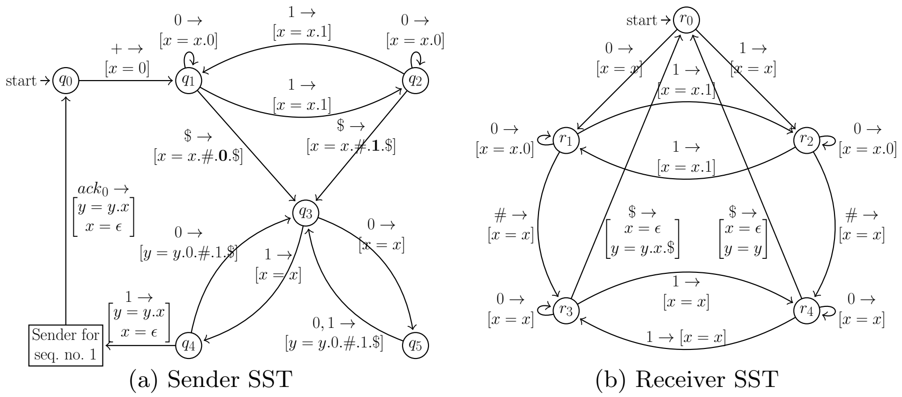

start
q_0
$+ \rightarrow [x = 0]$
q_1
q_2
q_3
q_4
q_5
$0 \rightarrow [x = x.0]$
$1 \rightarrow [x = x.1]$
$\$ \rightarrow [x = x.\#.\mathbf{0}.\$]$
$\$ \rightarrow [x = x.\#.\mathbf{1}.\$]$
$ack_0 \rightarrow \begin{bmatrix} y = y.x \\ x = \epsilon \end{bmatrix}$
$0 \rightarrow [y = y.0.\#.1.\$]$
$1 \rightarrow [x = x]$
$0 \rightarrow [x = x]$
$1 \rightarrow \begin{bmatrix} y = y.x \\ x = \epsilon \end{bmatrix}$
$0, 1 \rightarrow [y = y.0.\#.1.\$]$
Sender for seq. no. 1
(a) Sender SST
r_0
r_1
r_2
r_3
r_4
$0 \rightarrow [x = x]$
$1 \rightarrow [x = x]$
$1 \rightarrow [x = x.1]$
$0 \rightarrow [x = x.0]$
$\# \rightarrow [x = x]$
$\$ \rightarrow \begin{bmatrix} x = \epsilon \\ y = y.x.\$ \end{bmatrix}$
$\$ \rightarrow \begin{bmatrix} x = \epsilon \\ y = y \end{bmatrix}$
$1 \rightarrow [x = x]$
(b) Receiver SST

Fig. 5. SSTs for the TinyOS SerialP protocol with CRC polynomial $z + 1$

and is undefined for other states. Note that all the variable updates are copyless in both the SSTs.

Verification. We have implemented a prototype tool in OCaml to perform sequential composition and equivalence checking. For equivalence checking, our tool constructs the 1-counter automaton and uses ARMC for reachability checking [30]. The maximum time taken by sequential composition was 4s (for the HDLC protocol). Our implementation successfully verified both variants of TinyOS, but timed out on HDLC. The state space of 1-counter automata is exponential in the number of string variables. For HDLC, this proved to be a bottleneck. We aim to address scalability as part of the future work. One possibility is to explore minimization techniques for reducing the number of states and string variables of the protocol SST.

6 Related Work

The undecidability of verification problems for finite-state machines communicating asynchronously over unbounded *perfect* FIFO channels is shown in [12]. For some classes of systems and properties, decidability results are obtained in [19, 32]. For unbounded *lossy* FIFO channels, reachability, safety of system traces, and eventuality properties are decidable [4]. The semantics of lossy channels is orthogonal to our notion of noisy channels. For example, the scenario in Section 2 is applicable to lossy channels as well. A lossy forward channel may sometimes deliver corrupt messages to the receiver (and at other times, drop them). If we model messages as symbolic constants then the receiver cannot detect corruption and would deliver an incorrect sequence of messages to its client. Further, unlike these approaches, our choice of formalism for protocol modeling is transducers rather than communicating finite-state machines.

Sliding window protocols, being both complex and heavily used, are popular targets of verification techniques. Several automated techniques handle them by abstracting message contents. The work in [3] uses a class of regular expressions to represent channel contents where each message comes from a finite alphabet. In contrast, in our work, a message is treated as a finite bit string. The finite-state models obtained by abstracting message contents (and other parameters) are also verified by model checking (see survey [8]). Protocols with a fixed number of retransmission rounds are verified algorithmically in both untimed and timed cases in [17]. Our model does not impose bounds on the number of retransmission rounds. However, modeling timing constraints is beyond the scope of our method.

A number of deductive or semi-automated techniques have been developed for verification of sliding window protocols. The process-algebra framework LOTOS is used in [28], whereas, I/O automata are used with Coq theorem prover in [24]. Deductive theorem proving is used for reducing the complexity of protocol models and to obtain simpler abstractions suitable for algorithmic verification [23, 33]. Higher-order logic specifications of the protocols in the language of PVS are designed in [31]. Colored petri net models are used in [10] for modeling of stop-and-wait protocols. In [15], timed state machines are used for modeling sliding window protocols. Process algebra is also used in [9] to establish bi-similarity of these protocols with a queue.

The assumptions on reliability of channels vary across approaches. Similar to most of the above approaches, we consider non-duplicating FIFO channels. The approaches [33, 10] permit channels to re-order messages, whereas, [28, 15] permit both re-ordering and duplication. Most of the above approaches assume lossy channels and do not model message contents. Noisy channels are modeled in π-calculus through probabilistic semantics [38, 13]. A recent work [18] investigates decidability of control state reachability for ad-hoc networks in the presence of different types of node and communication failures.

Finite-state transducers are used in regular model checking for representing transition relations of systems whose configurations can be modeled as words [27, 37, 11]. The set of reachable states of these systems are represented by finite automata. However, in this context, the termination of fix-point computation is not guaranteed and the verification problem is in general undecidable. In contrast, the verification problem for the transducer models of the protocols presented by us, posed as functional equivalence checking, is decidable. Recently, more expressive transducer models are being designed by researchers, leading to new approaches to verification. SSTs have been introduced for pre/post verification and equivalence checking of single-pass programs operating over lists [6]. Symbolic finite transducers are introduced to analyze web sanitization functions [36].

7 Conclusions and Future Work

In this paper, we consider noisy communication channels that may corrupt messages. We show that for accurate modeling of the retransmission protocols, in the setting of noisy channels, the messages, checksums, and acknowledgements must be modeled at the bit-level. We propose streaming string transducers as a modeling framework for retransmission protocols. Even though the message lengths and retransmission rounds are unbounded, we present an algorithm to verify the protocol models. In future, we want to explore non-deterministic and bounded versions of the retransmission protocol models.

Acknowledgements. The first two authors were partially supported by Robert Bosch Centre for Cyber Physical Systems at the Indian Institute of Science.

References

1. `http://www.tinyos.net/tinyos-2.x/doc/html/tep113.html`
2. `http://www.ietf.org/rfc/rfc793.txt`
3. Abdulla, P.A., Annichini, A., Bouajjani, A.: Symbolic Verification of Lossy Channel Systems: Application to the Bounded Retransmission Protocol. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 208–222. Springer, Heidelberg (1999)
4. Abdulla, P.A., Jonsson, B.: Verifying Programs with Unreliable Channels. Inf. Comput. 127(2), 91–101 (1996)
5. Alur, R., Cerný, P.: Expressiveness of streaming string transducers. In: FSTTCS, pp. 1–12 (2010)
6. Alur, R., Cerný, P.: Streaming Transducers for Algorithmic Verification of Single-pass List-processing Programs. In: POPL, pp. 599–610 (2011)
7. Alur, R., Deshmukh, J.V.: Nondeterministic Streaming String Transducers. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 1–20. Springer, Heidelberg (2011)
8. Babich, F., Deotto, L.: Formal Methods for Specification and Analysis of Communication Protocols. IEEE Comm. Surveys and Tutorials 4(1), 2–20 (2002)
9. Badban, B., Fokkink, W., Groote, J., Pang, J., Pol, J.: Verification of a Sliding Window Protocol in μCRL and PVS. Formal Asp. Comput. 17(3), 342–388 (2005)
10. Billington, J., Gallasch, G.E.: How Stop and Wait Protocols Can Fail over the Internet. In: König, H., Heiner, M., Wolisz, A. (eds.) FORTE 2003. LNCS, vol. 2767, pp. 209–223. Springer, Heidelberg (2003)
11. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular Model Checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000)
12. Brand, D., Zafiropulo, P.: On Communicating Finite-State Machines. J. ACM 30(2), 323–342 (1983)
13. Cao, Y.: Reliability of Mobile Processes with Noisy Channels. IEEE Trans. Computers 61(9), 1217–1230 (2012)
14. Cerf, V., Kahn, R.: A Protocol for Packet Network Intercommunication. IEEE Transactions on Communications 22(5), 637–648 (1974)
15. Chkliaev, D., Hooman, J., de Vink, E.P.: Verification and Improvement of the Sliding Window Protocol. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 113–127. Springer, Heidelberg (2003)
16. Chytil, M., Jákl, V.: Serial composition of 2-way finite-state transducers and simple programs on strings. In: Salomaa, A., Steinby, M. (eds.) ICALP 1977. LNCS, vol. 52, pp. 135–147. Springer, Heidelberg (1977)

17. D'Argenio, P.R., Katoen, J.P., Ruys, T.C., Tretmans, G.J.: The Bounded Retransmission Protocol must be on time! In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 416–431. Springer, Heidelberg (1997)
18. Delzanno, G., Sangnier, A., Zavattaro, G.: Verification of Ad Hoc Networks with Node and Communication Failures. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 235–250. Springer, Heidelberg (2012)
19. Finkel, A.: Decidability of the termination problem for completely specified protocols. Distrib. Comput. 7(3), 129–135 (1994)
20. Forouzan, B.: Data Communications and Networking. McGraw-Hill Companies (2012)
21. Groote, J., Pol, J.: A Bounded Retransmission Protocol for Large Data Packets. In: Nivat, M., Wirsing, M. (eds.) AMAST 1996. LNCS, vol. 1101, pp. 536–550. Springer, Heidelberg (1996)
22. Gurari, E.: The equivalence problem for deterministic two-way sequential transducers is decidable. SIAM J. Comput. 11(3), 448–452 (1982)
23. Havelund, K., Shankar, N.: Experiments in Theorem Proving and Model Checking for Protocol Verification. In: Gaudel, M.-C., Wing, J.M. (eds.) FME 1996. LNCS, vol. 1051, pp. 662–681. Springer, Heidelberg (1996)
24. Helmink, L., Sellink, M.P.A., Vaandrager, F.W.: Proof-Checking a Data Link Protocol. In: Barendregt, H., Nipkow, T. (eds.) TYPES 1993. LNCS, vol. 806, pp. 127–165. Springer, Heidelberg (1994)
25. Holzmann, G.J.: The Model Checker SPIN. IEEE Trans. Software Eng. 23(5), 279–295 (1997)
26. ISO. Data Communication - HDLC Procedures - Elements of Procedure. Technical Report ISO 4335, International Organization for Standardization (1979)
27. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic Model Checking with Rich Assertional Languages. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 424–435. Springer, Heidelberg (1997)
28. Madelaine, E., Vergamini, D.: Specification and Verification of a Sliding Window Protocol in LOTOS. In: FORTE, pp. 495–510 (1991)
29. Peterson, W.W., Brown, D.T.: Cyclic Codes for Error Detection. In: IRE, pp. 228–235 (1961)
30. Podelski, A., Rybalchenko, A.: ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2007)
31. Rusu, V.: Verifying a Sliding Window Protocol using PVS. In: FORTE, pp. 251–268 (2001)
32. Sistla, A.P., Zuck, L.D.: Automatic Temporal Verification of Buffer Systems. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575, pp. 59–69. Springer, Heidelberg (1992)
33. Smith, M.A., Klarlund, N.: Verification of a Sliding Window Protocol Using IOA and MONA. In: FORTE, pp. 19–34 (2000)
34. Stenning, V.: A Data Transfer Protocol. Computer Networks 1, 99–110 (1976)
35. Tanenbaum, A.S., Wetherall, D.: Computer Networks. Pearson (2010)
36. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjørner, N.: Symbolic Finite State Transducers: Algorithms and Applications. In: POPL, pp. 137–150 (2012)
37. Wolper, P., Boigelot, B.: Verifying Systems with Infinite but Regular State Spaces. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 88–97. Springer, Heidelberg (1998)
38. Ying, M.: π-calculus with noisy channels. Acta Inf 41(9), 525–593 (2005)

Asynchronously Communicating Visibly Pushdown Systems

Domagoj Babić[1] and Zvonimir Rakamarić[2]

[1] Facebook, Inc., USA
babic.domagoj@gmail.com
[2] University of Utah, USA
zvonimir@cs.utah.edu

Abstract. We introduce an automata-based formal model suitable for specifying, modeling, analyzing, and verifying asynchronous task-based and message-passing programs. Our model consists of visibly pushdown automata communicating over unbounded reliable point-to-point first-in-first-out queues. Such a combination unifies two branches of research, one focused on task-based models, and the other on models of message-passing programs. Our model generalizes previously proposed models that have decidable reachability in several ways. Unlike task-based models of asynchronous programs, our model allows sending and receiving of messages even when stacks are not empty, without imposing restrictions on the number of context-switches or communication topology. Our model also generalizes the well-known communicating finite-state machines with recognizable channel property allowing (1) individual components to be visibly pushdown automata, which are more suitable for modeling (possibly recursive) programs, (2) the set of words (i.e., languages) of messages on queues to form a visibly pushdown language, which permits modeling of remote procedure calls and simple forms of counting, and (3) the relations formed by tuples of such languages to be synchronized, which permits modeling of complex interactions among processes. In spite of these generalizations, we prove that the composite configuration and control-state reachability are still decidable for our model.

1 Introduction

The asynchronous message-passing programming paradigm is becoming a de facto standard for parallel and distributed computing (e.g., cloud applications, web services, scientific computing). Programming such asynchronous systems is, however, difficult. In addition to having to reason about concurrency, programmers typically do not have full control over all the services they use. Therefore, failures are rarely reproducible, rendering debugging all but impossible. In response, programmers succumb to logging interesting events and gathering various statistics, hoping that if something goes wrong the logs will reveal the source of failure.

On the positive side, this is an opportunity for the scientific community to provide appropriate computationally tractable formal models, as well as programming paradigms, languages, and analysis tools based on such models. We propose such a formal model for asynchronous message-passing programs. The model generalizes several existing well-known models, but we prove that checking the system's safety properties is still decidable. More precisely, we propose an abstract automata-based model, in which individual processes are modeled by visibly pushdown automata (VPA) [2] that communicate via unbounded point-to-point reliable first-in-first-out (FIFO) queues. VPA are single-stack pushdown automata where all stack push and pop operations must be visible (i.e., explicit) in the input language. Such automata are commonly used to represent abstractions (e.g., computed using predicate abstraction [16,4]) of possibly recursive programs.

Unfortunately, reachability is undecidable even for finite-state machines communicating over unbounded queues (a.k.a. CFSMs) [10]. Researchers proposed a number of restrictions to regain decidability: bounding the size of queues to some fixed size, restricting the communication topology, and restricting the expressiveness of the languages representing the messages on queues. Pachl [22] proved that if a CFSM has a recognizable channel property — all the queue languages are regular and all those languages form a recognizable relation,[1] then reachability is decidable.

Pachl's restrictions are too restrictive in practice. Recognizable relations are a very inexpressive class of relations that can model inter-dependencies among queues only if languages describing the contents of each queue are finite. For instance, if we have an invariant that there should be the same number of messages, say a and b, on two queues in some composite control state, the relation representing the configuration of queues would be (a^n, b^n), which is not a recognizable relation. Even simple systems, like a client sending some number of requests and expecting the same number of responses, require queue relations that allow inter-dependencies (i.e., synchronization) among individual queue languages.

We relax Pachl's restrictions by allowing (but not requiring!) queue (and stack) configurations to form synchronized visibly pushdown relations, which significantly broadens the applicability of our model. Although in our model the two extensions, from regular to visibly pushdown languages and from recognizable to synchronized relations, go hand-in-hand, it is worth noting that they are orthogonal and each is valuable on its own. For instance, our relaxation from recognizable to synchronized relations is applicable to other models as well — a straightforward consequence of our results is that reachability of CFSMs with synchronized channel property is decidable.

The main technical contribution of this paper is a proof that model checking safety properties — global control state and global configuration reachability — is decidable for the model we propose. The result is non-trivial as the introduced model allows unbounded stacks and queues, arbitrary communication topologies, as well as complex inter-dependencies of queue and stack languages.

[1] Informally, a relation is recognizable if the concatenation of all the languages that are elements of the relation tuple is a regular language (see Sec. 3.3).

These extensions allow our model to capture the following distributed programming patterns:

Remote Procedure Calls. Processes can (recursively) call procedures to be executed on behalf of remote processes.

Message Counting. Processes can use their local stack to count the number of messages and assure that the number of responses matches the number of requests.

Asynchronous Communication. Processes send and receive messages asynchronously. While we present the model with FIFO channels, bag-like (i.e., multiset-like) queues can be simulated by receiving each message (assuming there are finitely many) on a separate queue. The receiver can non-deterministically choose which queue to process next. These features allow us to model both asynchronous communication often used for hiding latency in web services as well as non-deterministic interleaving of messages from different senders in publish-subscribe settings.

Synchronization. Often in practice, processes send multiple messages at once to different receivers. CFSMs with recognizable channel property are unable to model that behavior, as it creates dependencies among queues. Our generalization to synchronized relations allows us to handle non-trivial queue dependencies.

We summarize the contributions of this paper as follows:

– A new formal model for asynchronous message-passing programs. It is a relaxation of known communication restrictions along two dimensions: from regular to visibly pushdown languages, and from recognizable to synchronized relations.
– A proof of decidability of control state and configuration reachability in our model.

Our technical report [3] provides a more comprehensive coverage of this material.

2 Applications

In this section, we discuss the applications of the introduced model in the context of two major asynchronous programming paradigms: the task-based and the message-passing paradigm.

Asynchronous Task-Based Paradigm. The task-based programming paradigm enables programmers to break up time-consuming operations into a collection of shorter tasks. This adds reactivity to the system, and typically improves responsiveness and performance of long-running programs. Tasks can be either asynchronously posted for execution by other tasks, or triggered by events. These two approaches (and their combination) have been successfully employed in many domains: they form the basis of JavaScript and Silverlight (client-side) web applications, and have been shown useful for building fast servers [23], routers [19], and embedded sensor networks [18].

The formal system we propose can model this class of applications as follows. Each task (and there is a finite number of them) is executed on a single VPA. During execution, each task can change the state of its VPA and send messages to other VPAs, which is sufficient for modeling the global shared state changes that the task-based models can model. The task buffer is modeled as a FIFO queue: posting a task amounts to sending an invocation message to the task buffer queue.

Message-Passing Paradigm. The message-passing paradigm, in which processes communicate exclusively by sending messages to each other, has been implemented in a number of different ways: as an integral part of a programming language (e.g., Erlang, Scala), as a message-passing API implemented as a library (e.g., MPI, SOAP, Java Message Service, Microsoft Message Queuing), or as a software as a service model (e.g., Amazon Simple Queue Service). Message-passing applications can be viewed as a network of processes communicating over FIFO queues. It is straightforward to model such networks as a system of CVPTs: each (recursive) process can be abstracted into a Boolean program that sends and receives messages, and in turn the language of traces such program generates is accepted by a visibly pushdown transducer. For example, Erlang's message send and receive operations (i.e., `!` and `receive`) closely match send and receive operations in our model. It is also natural to map basic MPI asynchronous blocking send and receive operations (i.e., `MPI_Send` and `MPI_Recv`) to our model. Web services, another class of message-passing applications, are Internet-based applications that communicate and exchange data with other available web services in order to implement required functionality. The services typically communicate via asynchronous message-passing (e.g., SOAP, Ajax), and therefore again fit into our model.

3 Background and Related Work

The research on abstract models of asynchronous computation has progressed along two, mostly disjoint, paths. The first path stemmed from the classical negative result of Ramalingam [26] stating that the reachability for stack-based finite-data abstractions of concurrent programs with preemption is undecidable. Further research focused on computationally more tractable models, like context-bounded [24] and task-based non-preemptive models. The latter are more relevant to this paper. The second path originated in the study of finite-state machines communicating over reliable unbounded queues [10], known as CFSMs. Reachability is undecidable for general CFSMs. Further research focused on restrictions of CFSMs, especially of queue relations, and development of model checking algorithms exploiting such restrictions. In this paper, we unify the two paths, proposing a model suitable for the same class of applications as the task-based non-preemptive models, while significantly generalizing CFSMs with recognizable channel property, one of the more popular decidable restrictions. We proceed by surveying the related work along both paths, and then providing the necessary background on synchronized relations, which allow us to express complex inter-dependencies among queues and stacks.

3.1 Task-Based Models

The task-based model consists of a pushdown automaton and a task buffer for storing asynchronous task invocations. After the currently executing task returns, a scheduler takes another task from the task buffer and executes it on the automaton. Tasks execute atomically and can change the global state (of the automaton), but new tasks can start executing only when the stack is empty, i.e., the model is non-preemptive. Tasks cannot send messages to each other and communicate only by changing the global state. The model is suitable for modeling event-based applications (e.g., JavaScript programs) and simple worker-pool-based multithreaded applications (e.g., servers).

Sen and Viswanathan [28] proposed a task-based model where the task buffer is modeled as a multiset (i.e., a bag) and showed that the state exploration problem is EXPSPACE-hard. Ganty and Majumdar [14] did a comprehensive study of multiset task-based models, proving EXPSPACE-completeness of safety verification, and proposed a number of extensions. For instance, they show that the configuration reachability problem for the task-based model with task cancellation is undecidable. Our model does not allow message cancellation, but once the execution starts on some automaton, it is possible to send an abort message, which can change the course of execution.

La Torre et al. [20] studied a different set of trade-offs. Similarly to ours, their model allows unbounded reliable queues instead of multisets, but either bounds the number of allowed context-switches or restricts the communication topology to assure computational tractability. Similarly to the multiset-based model, their model can dequeue messages from queues only when the local stack is empty. Each VPA in our model can both send and receive messages, independently of the state of the local stack, as long as the languages of all the queues and stacks in the system can be described by a synchronized relation. Furthermore, we neither impose restrictions on the communication topology, nor require the number of context-switches to be bounded.

3.2 Communicating Finite-State Machines

Another line of research on formal models of asynchronous computation focused on CFSMs [10]. A CFSM is a system of finite-state machines operating in parallel and sending messages to each other via unbounded FIFO queues. CFSMs can model preemption, but finite-state machines are an overly coarse abstraction of (possibly recursive) programs. As discussed earlier, reachability is undecidable for CFSMs, in general. Basu et al. [5] show that a sub-class of asynchronous CFSMs can be encoded into synchronous systems, where reachability can be efficiently computed. They present an approach for deciding whether an asynchronous system belongs to that sub-class. Pachl [21,22] found that if the language of messages on each queue is regular and the tuple of such languages for all queues is a recognizable relation, then the reachability problem is decidable for CFSMs.

His work was followed by extensive research on, so called, regular model checking (e.g., [6,9,30,8]), where queue contents are described using recognizable relations over words. Model checking is then done by computing (sometimes approximations of) a transitive closure of the system's transition relation, and checking whether the image of the transitive closure is contained in the relation describing the queue contents. As the focus of this paper is on proposing a new formal model for modeling asynchronously communicating programs, and proving that the model has a decidable reachability problem, rather than on algorithms, we omit an extensive account of the regular model checking work. Instead, we direct an interested reader to a survey [1]. We suspect that techniques similar to the ones developed for regular model checking, especially to those for regular tree model checking (e.g., [7]), could be applied to model check the formal model we propose.

We generalize Pachl's results along two dimensions. First, the components of our formal model are visibly pushdown transducers, which can closely model the control flow of recursive programs. Therefore, they are a better candidate for modeling asynchronously communicating programs (e.g., event-based programs, web services, cloud applications, scientific computing applications) than the less powerful finite-state machines. Accordingly, we allow queue relations in our model to be visibly pushdown [2], rather than just regular. This relaxation enables us to support remote procedure calls and limited forms of unbounded message counting. Second, we significantly relax the restrictions on queue relations, allowing more expressive communication patterns. More precisely, we show that our model has a decidable reachability problem even when we move one step up in the hierarchy of families of relations from the family used by Pachl. Such more expressive relations allow us to model complex inter-dependencies among queue and stack configurations.

3.3 Relations over Words and Trees

In this section, we give an overview of the main results on relations over regular sets (of words and trees) relevant to this paper. The properties of those relations are the key to understanding the presented contributions. A property that we are particularly interested in is the decidability of language inclusion ($\subseteq$), which we use in the proof of the decidability of reachability in our formal model. We start with the least expressive family of relations (see Fig. 1).

$$\begin{array}{ccccccc} \checkmark & & \checkmark & & \oslash & & \oslash \\ \mathit{Rec} & \subset & \mathit{Sync} & \subset & \mathit{DRat} & \subset & \mathit{Rat} \\ [29] & & [12] & & [25] & & [25] \end{array}$$

Fig. 1. Hierarchy of Relations Over Words and Trees. The checkmark (✓) denotes the relations for which inclusion is decidable. Such relations when used to describe the queue and stack languages in our model maintain the decidability of reachability.

Recognizable relations (*Rec*) [29] have the weakest expressive power of all families we discuss. Each tape of an n-tape automaton operates independently of others and has its own memory. The relations accepted by such automata can be represented as finite unions of cross-products of regular component languages. Effectively, the component languages can be all concatenated together and recognized by a 1-tape automaton. Pachl's work, as well as most work on regular model checking, focuses on this family of relations, which are insufficiently expressive to describe complex inter-dependencies among queues.

Synchronized relations (*Sync*) [12] are strictly more expressive than recognizable relations. Synchronized n-tape automata over $\Sigma^* \times \cdots \times \Sigma^*$ can be seen as the classical 1-tape automata over the alphabet that is a cross-product of alphabets of all tapes, i.e., $(\Sigma \times \cdots \times \Sigma)^*$. Such automata move all their tape heads synchronously in lock-step, as if it is a single head reading a tuple of symbols. Synchronized relations are sufficiently expressive to describe languages such as (a^m, b^m), which is useful for describing asynchronously communicating programs in which processes can send multiple messages to different queues at the same time. By adding a special padding symbol (#), synchronized relations can also be used to describe languages such as (a^m, b^k), where $k > m$. Synchronized relations have essentially the same properties as the 1-tape automata (closure under union, intersection, etc.) and their inclusion can be efficiently checked.

Frougny and Sakarovitch [13] defined *resynchronizable relations*, which describe languages of n-tape automata whose tapes are not synchronized, but the distance between tape heads is a-priori bounded. Such relations can be characterized as a finite union of the component-wise products of synchronized relations by finite sets, which in turn means they can be reduced to synchronized relations. For instance, $(b^m aab^k, c^m d^k)$ is an example of a resynchronizable relation: after reading (b^m, c^m), the first tape reads two more symbols (increasing the distance between tape heads to two), and then both tapes can again move together in sync. Relations like $((a^*b)^m, c^m)$ are not resynchronizable, as the distance between tape heads can become arbitrarily large. Our proof technique is applicable to all families of relations reducible to synchronized relations.

Rabin and Scott [25] introduced a generalization of the finite automata operating on words (single tape) to tuples of words (multiple tapes). Such automata realize regular transductions. The basic variants of such automata are non-deterministic and deterministic, accepting *rational relations* (*Rat*) and *deterministic rational relations* (*DRat*), respectively. While the equivalence problem of *DRat* is decidable [17], inclusion is unfortunately undecidable for both classes. Therefore, our proof technique cannot be used to prove decidability of reachability in systems of CFSM or visibly pushdown transducers whose queue languages form (deterministic) rational relations.

4 The Formal Model

In this section, we describe our formal model. We begin by describing the basic component — a visibly pushdown transducer, continue with a definition of a system of such transducers communicating over reliable unbounded queues, and finish with a discussion of relations describing queue and stack configurations.

Notations and Terminology. We define *disjoint union* $A \uplus B$ as the standard set union $A \cup B$, but with an implicit side-constraint that the sets A and B are disjoint. Let I be a set. An *I-indexed* set A is defined as a disjoint union of sets indexed by elements of I, i.e., $A = \biguplus_{i \in I} A_i$. We denote tuples by a vector sign, e.g., $\overrightarrow{t}$. If $\overrightarrow{t} = (obj_1, \ldots, obj_n)$ is a tuple, n is called the *size* of the tuple and denoted $|\overrightarrow{t}|$. The *cross-product* of sets $A_1, \ldots, A_n$, denoted

$\prod_{1 \le i \le n} A_i$, is a set of n-tuples $\{(a_1, \ldots, a_n) \mid a_i \in A_i\}$. The i-th element of a tuple $\vec{t}$ is denoted $\vec{t}|_i$. We write S^n for a set of n-tuples in which all elements are from S. Let $\vec{u}$ and $\vec{v}$ be tuples of words such that $|\vec{u}| = |\vec{v}|$; the *component-wise product* of $\vec{u}$ and $\vec{v}$, denoted $\vec{u} \cdot \vec{v}$, is a tuple of words $\vec{t}$ such that for $1 \le i \le |\vec{u}|$, $\vec{t}|_i = \vec{u}|_i \cdot \vec{v}|_i$. The *power of a word* w is defined recursively: $w^0 = \epsilon$, $w^{k+1} = w \cdot w^k$. If A, B are languages, then their concatenation $A \cdot B$ is the language $\{u \cdot v \mid u \in A, v \in B\}$. If w is a word and A is a language, then $w \cdot A = \{w \cdot u \mid u \in A\}$, $A \cdot w = \{u \cdot w \mid u \in A\}$. *(Left) language quotient* $a^{-1}A$ is the language $\{u \mid a \cdot u \in A\}$. Let $u, v \in \Sigma^*$ be words over Σ. The *prefix order* $\le$ is defined as: $u \le v$ iff there exists $w \in \Sigma^*$ such that $v = u \cdot w$. We say that a set S is *prefix-closed* if $u \le v \wedge v \in S \Rightarrow u \in S$.

4.1 Visibly Pushdown Transducers

The individual processes receive and process words of input messages, and generate words of output messages. Thus, they can be modeled as transducers — state machines translating one language into another. We introduce such a machine with a finite-state control, a single stack, and a finite set of unbounded FIFO queues. On each step it can read a symbol from one queue and write to another; if the symbol read is a call (resp. return), it can simultaneously push to (resp. pop from) its stack.

Definition 1. *A* communicating visibly pushdown transducer *(CVPT) is a tuple $T = (\Sigma_{rcv}, \Sigma_{snd}, Q, S, I, F, \Gamma, \Delta)$ of finite sets, where Σ_{rcv} is an input alphabet, Σ_{snd} an output alphabet, Q a set of unbounded FIFO queues, S a set of states, $I \subseteq S$ a set of initial states, $F \subseteq S$ a set of final states, Γ an alphabet of stack symbols, and Δ a transition relation. The input alphabet Σ_{rcv} and the output alphabet Σ_{snd} are disjoint sets indexed by Q, which means that the set of messages that can be sent to any $q_i \in Q$ is disjoint from messages that can be sent to all other queues $Q \setminus q_i$. Another way to partition the input alphabet is $\Sigma_{rcv} = \Sigma_c \uplus \Sigma_r \uplus \Sigma_i$, where Σ_c is an alphabet of* call symbols*, Σ_r an alphabet of* return symbols*, and Σ_i the* internal *alphabet. For each return in Σ_r there exists a matching call in Σ_c, more formally: $\Sigma_r = \{\underline{c} \mid \overline{c} \in \Sigma_c\}$ and $|\Sigma_r| = |\Sigma_c|$.*[2] *The set of queues Q contains a special symbol $\perp \in Q$ used in transitions that do not receive input from (or send output to) a queue.*

A *configuration C of a CVPT* is a tuple $(s, \sigma, \vec{\varrho}) = (s, \sigma, \varrho_1, \ldots, \varrho_{|Q|}) \in S \times \Gamma^* \times \prod_{q \in Q} (\Sigma_{snd_q} \cup \Sigma_{rcv_q})^*$, representing a control state, a word on the stack, and contents (represented as words) of each of CVPT's queues. For stacks, the leftmost symbol of the word is the top of the stack. For queues, the leftmost symbol of the word represents the next message to be processed (i.e., the oldest yet unprocessed message), while the rightmost symbol represents the most recently received message. To simplify the notation, we assume that ϱ_i represents the contents of queue $q_i \in Q$ and ϱ the contents of queue q. We use the

[2] We denote call symbols with overline (e.g., $\overline{c}$), and return symbols with underline (e.g., $\underline{c}$).

$C[oldstate \leftarrow newstate]$ parallel substitution notation to represent incremental modifications of configurations. For example, $C[s_1 \leftarrow s_2, \sigma \leftarrow a \cdot \sigma, \varrho_3 \leftarrow \varrho_3 \cdot b]$ denotes a configuration C modified so that the control state is changed from s_1 to s_2, message a is pushed on the stack, and message b is appended to queue q_3; $C[m \cdot \varrho \leftarrow \varrho]$ denotes a configuration C modified so that message m is removed from queue q. We define the transition relation of a CVPT as follows.

Definition 2 (CVPT Transition Relation). *Let C be a configuration of a CVPT T. If $m \in \Sigma_{snd_q}$, for some $q \in Q$, let $q!m$ (resp. $q?m$) be an alias name for message m sent to (resp. received from) queue q.*[3] *If $m = \epsilon$, then $q = \perp$. The transition relation $\Delta = \delta_c \uplus \delta_r \uplus \delta_i$, such that $\delta_c \subseteq S \times \Sigma_c \times (\Sigma_{snd} \cup \{\epsilon\}) \times \Gamma \times S$, $\delta_r \subseteq S \times \Sigma_r \times \Gamma \times (\Sigma_{snd} \cup \{\epsilon\}) \times S$, and $\delta_i \subseteq S \times (\Sigma_i \cup \{\epsilon\}) \times (\Sigma_{snd} \cup \{\epsilon\}) \times S$, is defined as follows ($\xrightarrow[x]{}$ is the infix notation for δ_x):*

Call. *If $(s_1, q_1?\overline{m}_1, q_2!m_2, \gamma, s_2) \in \delta_c$, then $C \xrightarrow[c]{q_1?\overline{m}_1/q_2!m_2,\gamma} C[s_1 \leftarrow s_2, \sigma \leftarrow \gamma \cdot \sigma, \overline{m}_1 \cdot \varrho_1 \leftarrow \varrho_1, \varrho_2 \leftarrow \varrho_2 \cdot m_2]$;*

Return. *If $(s_1, q_1?\underline{m}_1, \gamma, q_2!m_2, s_2) \in \delta_r$, then $C \xrightarrow[r]{q_1?\underline{m}_1,\gamma/q_2!m_2} C[s_1 \leftarrow s_2, \gamma \cdot \sigma \leftarrow \sigma, \underline{m}_1 \cdot \varrho_1 \leftarrow \varrho_1, \varrho_2 \leftarrow \varrho_2 \cdot m_2]$;*

Internal. *If $(s_1, q_1?m_1, q_2!m_2, s_2) \in \delta_i$, then $C \xrightarrow[i]{q_1?m_1/q_2!m_2} C[s_1 \leftarrow s_2, m_1 \cdot \varrho_1 \leftarrow \varrho_1, \varrho_2 \leftarrow \varrho_2 \cdot m_2]$.*

When we do not care about the exact type of a transition, we use $\longrightarrow$ to represent any of the $\xrightarrow[x]{}$ transitions defined above. A *run* of a CVPT on a word $w = a_0 \cdots a_n \in \Sigma_{rcv}^*$ from a configuration C is a finite sequence of configurations $C_0, C_1, \ldots, C_n$, such that $C_0 = C$ and for each $0 < i \leq n$ there exist a transition $C_{i-1} \longrightarrow C_i$. An *accepting run* is a run in which $C_n = (s_n, \sigma_n, \vec{\varrho}_n)$ and $s_n \in F$. A word $w \in \Sigma_{rcv}^*$ is accepted by a CVPT A if there is an accepting run of A on w. The language of A, denoted $\mathcal{L}(A)$, is the set of words accepted by A. We extend the infix notation, defined above for transitions, to words: $C \xrightarrow{w/p} C'$ if there exists a run on w from C to C' yielding output word p. When we are interested only in the input word, say w, we omit the output word, e.g., $C_1 \xrightarrow{w} C_2$. The transitive closure of the $\longrightarrow$ relation is denoted $\xrightarrow{*}$. Let $[\![T]\!]$ be the transduction induced by T: if there is a run $(s_0, \epsilon, \vec{\epsilon}) \xrightarrow{w/p} (s, \sigma, \vec{\varrho})$, where $s_0 \in I$ and $\vec{\epsilon}$ is a tuple of empty strings, then $p \in [\![T]\!](w)$. We generalize the transduction $[\![T]\!]$ to languages as usual, i.e., $[\![T]\!](L) = \{[\![T]\!](w) \mid w \in L\}$.

We use Definition 1 for two purposes. First, we use it to define individual components of a system of asynchronously communicating processes. The set of final states could be empty for such components, if we are interested in the computation those components perform, rather than the language they accept. Second, we use Definition 1 to define visibly pushdown languages (VPLs), which in turn we use to define conditions under which reachability is still decidable for

[3] Note that $m = q!m = q?m$ for any q and m. The alias names are just a notational convenience.

our model. When defining VPLs, the set of final states will be non-empty, but the output alphabet Σ_{snd} will be empty.

Definition 3. *A CVPT with* $\Sigma_{snd} = \emptyset$ *is a* visibly pushdown automaton *(VPA). A language of finite words* $L \subseteq \Sigma_{rcv}^*$ *is a* visibly pushdown language *(VPL) if there exists a VPA* A *over* Σ_{rcv} *accepting the language, i.e., if* $\exists A.\mathcal{L}(A) = L$*. Let* $\mathcal{V}$ *be the set of all VPL languages.*

We now sketch how to model a (possibly recursive) Boolean program P (i.e., a program with a finite number of variables over a finite domain) as a CVPT T. We choose a suitable alphabet of call, return, and internal symbols for representing statements of P. Then, every call statement of P is mapped into a $\underset{c}{\rightarrow}$ transition of T, and every return statement into a $\underset{r}{\rightarrow}$ transition. All other statements are mapped into simple internal transitions. Furthermore, the model could be extended with pre-initialized queues, with only one receiver and no senders, initialized to the language describing the program to be executed on the receiving automaton. However, such extensions significantly complicate the exposition, without contributing to the expressiveness of our model.

4.2 Systems of CVPTs

We compose CVPTs into more complex systems as follows.

Definition 4 (Asynchronous System of CVPTs). *An* asynchronous system of CVPTs $M = (T_1, \dots, T_n)$*, where* $T_i = (\Sigma_{rcv_i}, \Sigma_{snd_i}, \ Q_i, S_i, I_i, F_i, \Gamma_i, \Delta_i)$*, is a tuple of CVPTs, such that each FIFO queue has exactly one receiver and one sender. Any pair of CVPTs, say* T_j *and* T_k*, can share one or more queues* $q \in \bigcup_{1 \le i \le n} Q_i$ *such that sender's* $\Sigma_{snd_{j_q}}$ *is equivalent*[4] *to receiver's* $\Sigma_{rcv_{k_q}}$*.*

Since each queue in the system has a single receiver, we introduce a convention to avoid redundancy in specifying contents of the same queue: when we refer to a CVPT T_i as a part of a system, we consider that $\vec{\varrho}$ in T_i's configuration $(s, \sigma, \vec{\varrho})$ represents only the contents of T_i's input queues, i.e., the queues are considered to belong to the receiver.

A *composite configuration* is a tuple $\vec{C} = (C_1, \dots, C_n)$. Let $C_i = (s_i, \sigma_i, \vec{\varrho}_i)$ represent the configuration of the i-th CVPT in the system. We define the *composite control state* $\vec{s}$ of a system as a tuple of states $(s_1, \dots, s_n)$, *composite stack configuration* $\vec{\sigma}$ as a tuple of words $(\sigma_1, \dots, \sigma_n)$, and *composite queue configuration* $\vec{\varrho}$ as a tuple of words $(\varrho_{11}, \dots, \varrho_{1m_1}, \varrho_{21}, \ \dots, \varrho_{2m_2}, \dots, \varrho_{n1}, \dots, \varrho_{nm_n})$, where $m_i = |\vec{\varrho}_i|$ and $\varrho_{ij} = \vec{\varrho}_i|_j$. For a configuration $\vec{C}$, we write $\vec{C}.s$, $\vec{C}.\sigma$, and $\vec{C}.\varrho$ for the composite control state $\vec{s}$, composite stack configuration $\vec{\sigma}$, and composite queue configuration $\vec{\varrho}$.

We define the transition relation of a system in terms of transition relations of its individual components. Let $\vec{C}_0 = \prod_{1 \le i \le n} (s_i, \vec{\epsilon}, \vec{\epsilon})$, where $s_i \in I_i$, be an initial

[4] Note we index Σ in three different ways: Σ_i is the alphabet of T_i, Σ_q is the alphabet of the messages on queue q, and Σ_{i_q} is T_i's alphabet projected on the set of messages allowed on q.

composite configuration. A *run of a system* is a finite sequence $\vec{C}_0 \longrightarrow \vec{C}_1 \longrightarrow \cdots \longrightarrow \vec{C}_k$, where $\longrightarrow$ is defined as in Definition 2, with a minor difference that the output queues belong to another component, and not the one making the transition.

Now, we have all the formal machinery needed to define the configuration reachability problem for a system of CVPTs. Further discussion will focus on the composite configuration reachability, but we show later that our results can be somewhat generalized (e.g., to the composite control state reachability problem).

Problem 1. For a given composite configuration $\vec{C}$ of an asynchronous system M of CVPTs, does there exist a run of M ending in $\vec{C}$?

4.3 Relations Describing Configurations

For a given composite state $\vec{s}$ and a particular queue (resp. stack), we refer to the set of all words over messages (resp. stack symbols) describing the possible queue (resp. stack) contents in $\vec{s}$ as a queue (resp. stack) language. To define relations among those languages, we introduce stack and queue relations:

Definition 5 (Stack and Queue Relations). *Let $M = (T_1, \ldots, T_n)$ be a system of CVPTs. Let $\vec{C}_0$ be an initial composite configuration. We define the* queue relation $\mathbf{L}_q \subseteq \prod_{1\le i\le n} S_i \times \prod_{q\in Q} \Sigma^*_{rcv_q}$ *as* $\mathbf{L}_q(\vec{s}) = \left\{\vec{C}.\varrho \mid \vec{C}_0 \xrightarrow{*} \vec{C} \wedge \vec{C}.s = \vec{s}\right\}$, *and the* queue-stack relation $\mathbf{L}_{qs} \subseteq \prod_{1\le i\le n} S_i \times \prod_{q\in Q} \Sigma^*_{rcv_q} \times \prod_{1\le i\le n} \Gamma_i^*$ *as* $\mathbf{L}_{qs}(\vec{s}) = \left\{\vec{C}.\varrho, \vec{C}.\sigma \mid \vec{C}_0 \xrightarrow{*} \vec{C} \wedge \vec{C}.s = \vec{s}\right\}$, *where $Q = \bigcup_{1\le i\le n} Q_i$ is the set of all queues in M.*

In the next section, we introduce a family of synchronized tree relations, which we use to relax Pachl's restrictions (Sec. 3.2). Later, we prove that Problem 1 is decidable, despite our relaxations.

5 Tree Relations

In this section, we first develop a connection between VPLs and regular tree languages, building on top of prior work by Alur and Madhusudan [2]. We then define synchronized tree relations, using the appropriate encoding operator [11, p. 75].

5.1 Isomorphism between VPLs and Stack-Tree Languages

VPLs can be characterized in terms of, so called, stack-tree languages. We use this characterization to define the relations we are interest in. We start by defining trees and then develop the connection to VPLs.

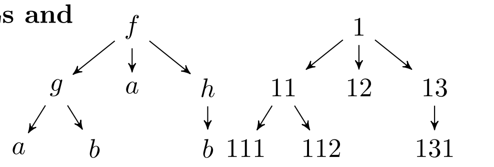


Fig. 2. An Example of a Tree t and its Tree Domain. $D = \{1, 11, 111, 112, 12, 13, 131\}$, $\mathcal{F} = \{f, g, h, a, b\}$, $\| t \| = 3$, $t(1) = f$.

Definition 6 (Trees). *Let $\mathbb{N}$ be the set of natural numbers. A* tree domain *is a finite non-empty prefix-closed set $D \subset \mathbb{N}^*$ satisfying the following property: if $u \cdot n \in D$ then $\forall 1 \leq j \leq n \,.\, u \cdot j \in D$. A* ranked alphabet *is a finite set $\mathcal{F}$ associated with a finite* ranking relation *$arity \subseteq \mathcal{F} \times \mathbb{N}$. Define $\mathcal{F}_n$ as a set $\{f \in \mathcal{F} | \, (f, n) \in arity\}$. The set $\mathbb{T}(\mathcal{F})$ of* terms *over the ranked alphabet $\mathcal{F}$ is the smallest set defined by:*

1. $\mathcal{F}_0 \subseteq \mathbb{T}(\mathcal{F})$;

2. if $n \geq 1,\ f \in \mathcal{F}_n,\ t_1, \ldots, t_n \in \mathbb{T}(\mathcal{F})$ then $f(t_1, \ldots, t_n) \in \mathbb{T}(\mathcal{F})$.

Each term can be represented as a finite ordered tree *$t : D \to \mathcal{F}$, which is a mapping from a tree domain into the ranked alphabet such that $\forall u \in D$:*

1. if $t(u) \in \mathcal{F}_n,\ n \geq 1$ then $\{j \mid u \cdot j \in D\} = \{1, \ldots, n\}$;

2. if $t(u) \in \mathcal{F}_0$ then $\{j \mid u \cdot j \in D\} = \emptyset$.

The height *$\| \, t \, \|$ of a tree $t = f(t_1, \ldots, t_k)$ is the number of symbols along the longest branch in the tree, i.e., $max\,(\| \, t_1 \, \|, \ldots, \| \, t_k \, \|) + 1$.*

Fig. 2 shows an example of a tree and its tree domain.

Given a word $v \in \Sigma^*_{rcv}$, we say that v has *matched returns* (resp. *calls*) if it is a production of the grammar $W ::= a \mid W \cdot W \mid V \mid U,\ \ V ::= a \mid V \cdot V \mid \overline{c} \cdot V \cdot \underline{c}$ such that $U ::= \overline{c}$ (resp. $U ::= \underline{c}$), where $a \in \Sigma_i$, $\overline{c} \in \Sigma_c$, $\underline{c} \in \Sigma_r$. A word is *well-matched* if it has both matched returns and calls. Following Alur and Madhusudan [2], we define an injective map $\eta : \Sigma^*_{rcv} \to \mathbb{T}(\mathcal{F})$, illustrated in Fig. 3, that translates VPL words to *stack-trees* as follows:

$$
\begin{aligned}
&\eta(\epsilon) = \#;\\
&\eta(\overline{c}w) = \overline{c}(\eta(w), \#),\ \text{if there is no return } \underline{c} \text{ matching } \overline{c} \text{ in } w;\\
&\eta(\overline{c}w\underline{c}w') = \overline{c}(\eta(w), \eta(\underline{c}w')),\ \text{assuming } w \text{ is well-matched};\\
&\eta(aw) = a(\eta(w)),\ \text{if } a \in \Sigma_i \cup \Sigma_r.
\end{aligned}
$$

The ranked alphabet $\mathcal{F} = \mathcal{F}_0 \uplus \mathcal{F}_1 \uplus \mathcal{F}_2$ used in the translation is defined as follows: $\mathcal{F}_0 = \{\#\}$, where $\#$ is a special symbol, $\mathcal{F}_1 = \Sigma_i \cup \Sigma_r$, and $\mathcal{F}_2 = \Sigma_c$. Regular sets of stack-trees form stack-tree languages, which are isomorphic to VPLs [2].

We use the $\eta(\Sigma^*_{rcv})$ isomorphism to define, indirectly, VPL relations. Such relations can, broadly, be classified into those recognizable by various types of finite automata and those that are not recognizable. For instance, (a^n, b^{2^n}) is an example of a relation not recognizable by any finite-state machine. The *Rec* class of recognizable relations, introduced in Sec. 3.3, can be extended to regular (stack-) tree languages, in which case it correspond to relations that are finite unions of cross-products of regular (stack-) tree languages, denoted $Rec^{\mathcal{V}}$. The $Rec^{\mathcal{V}}$ class is recognizable by a tree automaton, but is insufficiently expressive. In particular, the languages that are elements of the cross-product are independent and cannot express relations like (a^n, b^n). This means that if we restricted the cross-product of queue languages to belong to $Rec^{\mathcal{V}}$, we could not express protocols that send n messages

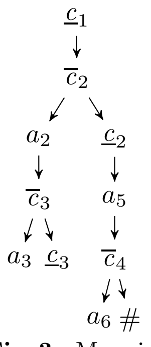


Fig. 3. Mapping η for a Word: $\underline{c}_1 \cdot \overline{c}_2 \cdot a_2 \cdot \overline{c}_3 \cdot a_3 \cdot \underline{c}_3 \cdot \underline{c}_2 \cdot a_5 \cdot \overline{c}_4 \cdot a_6$

(say a) asynchronously and than expect the same number of acknowledgments (say b). In other words, $Rec^{\mathcal{V}}$ does not allow us to express even simple forms of counting and synchronization.

5.2 Synchronized Tree Relations

We define a more expressive class of recognizable relations using *overlap encoding* [11, p. 75], inductively defined for a sequence of binary trees from $\mathbb{T}(\mathcal{F})$ as

$$[t_1, .., t_n] = \begin{cases} t_1(1) \cdots t_n(1) & \text{if } arity\,(t_i(1)) = 0 \\ t_1(1) \cdots t_n(1)\,([t_1(11), .., t_n(11)]) & \text{if } arity\,(t_i(1)) \leq 1 \\ t_1(1) \cdots t_n(1)\,([t_1(11), .., t_n(11)]\,, & \quad\text{otherwise} \\ \ [t_1(12), .., t_n(12)]) & \end{cases}$$

where $t_i(1k)$ is equal to $\#$ if $k > arity\,(t_i(1))$. An example of the overlap encoding is shown in Fig. 4. Using the notion of the overlap encoding, we can define synchronized tree relations as follows:

Definition 7 (Synchronized Tree Relations). *$Sync^{\mathcal{V}}$ is a family of relations $R \subseteq \mathbb{T}(\mathcal{F} \cup \{\#\})^n$ such that $\{[t_1, \ldots, t_n] \mid (t_1, \ldots, t_n) \in R\}$ is recognized by a finite tree automaton over the alphabet $(\mathcal{F} \cup \{\#\})^n$.*

$Sync^{\mathcal{V}}$ inherits all the properties from regular tree languages: it is closed under Boolean operations and both the equality and containment are decidable. Furthermore, $Sync^{\mathcal{V}}$ is known to be a strict superclass of $Rec^{\mathcal{V}}$ [11, p. 79] and allows us to express limited forms of counting, e.g., $(a^n, b^n) \in Sync^{\mathcal{V}}$. We use the introduced family of relations, $Sync^{\mathcal{V}}$, to define sufficient conditions for the decidability of reachability for a system of CVPTs in the next section.

6 Decidability of Reachability

In this section, we state and prove the main result of this paper. We begin by introducing sufficient conditions for decidability of reachability of a system of asynchronously communicating CVPTs, state the main theorem in Sec. 6.1, and prove it in the end.

6.1 Sufficient Conditions for the Decidability of Reachability

As discussed in Sec. 3.2, reachability is undecidable even for CFSMs. However, if relations representing queue configurations are restricted to regular and recognizable, reachability is decidable. In this section, we relax those restrictions, while maintaining decidability. First, we allow the languages representing contents of each queue to be visibly pushdown, rather than just regular. We require CVPTs not to generate context-free outputs, to assure that CVPTs in a system are composable. Second, we allow relations to be synchronized, rather than just recognizable. These relaxations are orthogonal and each is valuable on its own, but the combination is, of course, more powerful.

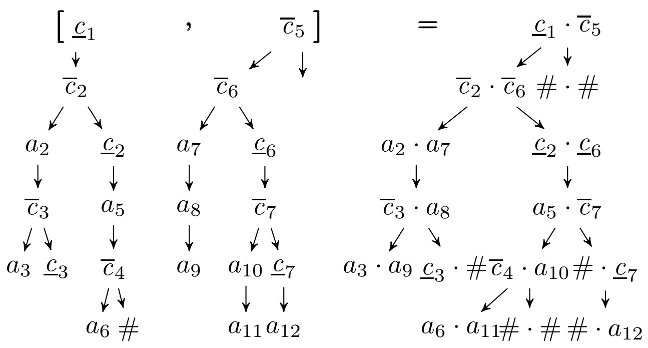


Fig. 4. An Example of the Overlap Encoding. The left (resp. middle) tree represents the $\underline{c}_1 \cdot \overline{c}_2 \cdot a_2 \cdot \overline{c}_3 \cdot a_3 \cdot \underline{c}_3 \cdot \underline{c}_2 \cdot a_5 \cdot \overline{c}_4 \cdot a_6$ (resp. $\overline{c}_5 \cdot \overline{c}_6 \cdot a_7 \cdot a_8 \cdot a_9 \cdot \underline{c}_6 \cdot \overline{c}_7 \cdot a_{10} \cdot a_{11} \cdot \underline{c}_7 \cdot a_{12}$) VPL word.

CVPT Composition. CVPTs are, in general, not closed under composition. As defined in Definition 1, CVPTs accept exactly VPLs. However, even for VPL inputs, CVPTs can generate context-free outputs [27]. As context-free relations do not have the properties we require (e.g., containment is undecidable), we introduce the following requirement:

Property 1 (Composition Property). Let $\pi_X : \Sigma^* \to X^*$ be a projection operator that erases all symbols from a word that are not in set X. For instance, if $X = \{a, b\}$ then $\pi_X(a \cdot d \cdot d \cdot b \cdot d) = a \cdot b$. Let $M = (T_1, \ldots, T_n)$ be a system of CVPTs. A CVPT T_j is said to be *composable* if a projection of its output language (i.e., a transduction of some VPL L) onto the input alphabet of any T_i is a VPL. More formally: $\forall 1 \leq i \leq n \,.\, L \in \mathcal{V} \implies \pi_{\Sigma_{rcv_i}}(\llbracket T_j \rrbracket(L)) \in \mathcal{V}$.

To understand the property better, suppose G is a graph representing a system M, such that vertices represent component CVPTs and edges represent communication between CVPTs; there is a directed edge between two nodes T_i and T_j if T_i sends messages to T_j. The above property assures that a non-VPL will never be generated on any path in G. Further on, we shall assume that all CVPTs have the composition property.

Synchronization. According to Property 1, the language representing the contents of each queue is a VPL. Thus, the contents of queues can be described by a cross-product of VPLs in every composite control state. Such VPL relations can be recognizable, synchronized, or rational (see Fig. 1). We use the concept of synchronized tree relations, introduced in Sec. 5.2, to define synchronized VPL relations. As we prove later, if queue and stack relations in every reachable composite control state are synchronized VPL relations, then the reachability is decidable for our model.

Property 2 (Synchronized Configuration Property). We say that an asynchronous system of CVPTs has the *synchronized configuration property* iff in every composite control state $\vec{s}$ reachable from an initial configuration $\vec{C}_0$, the encoding

$[\eta(\mathbf{L}_{qs}(\vec{s}))]$ is a synchronized tree relation, i.e., $\{[\eta(\sigma_1), \ldots, \eta(\sigma_n), \eta(\varrho_1), \ldots, \eta(\varrho_k)] \mid (\sigma_1, \ldots, \sigma_n, \varrho_1, \ldots, \varrho_k) \in \mathbf{L}_{qs}(\vec{s})\} \in Sync^{\mathcal{V}}$.

We now state the main result of this paper:

Theorem 1. *Reachability is decidable for a system of composable CVPTs with the synchronized configuration property.*

For the proof, please see our technical report [3].

7 Conclusions

In this paper, we proposed a new formal model for asynchronously communicating message-passing programs. The model is composed of visibly pushdown transducers communicating over unbounded reliable point-to-point FIFO queues. The proposed model is intended for specifying, modeling, analysis, and verifying of asynchronous message-passing programs and makes it possible to model (possibly recursive) programs and complex communication patterns. Our results generalize the prior work on communicating finite state machines along two directions — by allowing visibly pushdown languages on queues, and by allowing complex inter-dependencies (i.e., synchronization) among stack and queue languages. Our work also unifies two branches of research — one focused on task-based and the other on queue-based message-passing models. The results are non-trivial, because there are two sources of infiniteness: stacks and queues.

Acknowledgment. We would like to thank Brad Bingham, Jesse Bingham, Matko Botinčan, Steven McCamant, Jan Pachl, Shaz Qadeer, and Serdar Tasiran for their feedback on the early drafts of this document. The first author did this work while he was at UC Berkeley, where his research was sponsored by LLNL.

References

1. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004)
2. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Annual ACM Symp. on Theory of Computing (STOC), pp. 202–211 (2004)
3. Babić, D., Rakamarić, Z.: Asynchronously communicating visibly pushdown systems. Technical Report UCB/EECS-2011-108, University of California, Berkeley (October 2011)
4. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Conf. on Programming Language Design and Implementation (PLDI), pp. 203–213 (2001)
5. Basu, S., Bultan, T., Ouederni, M.: Synchronizability for verification of asynchronously communicating systems. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 56–71. Springer, Heidelberg (2012)

6. Boigelot, B., Godefroid, P., Willems, B., Wolper, P.: The power of QDDs. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 172–186. Springer, Heidelberg (1997)
7. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking. Electronic Notes in Theoretical Computer Science 149, 37–48 (2006)
8. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 372–386. Springer, Heidelberg (2004)
9. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000)
10. Brand, D., Zafiropulo, P.: On communicating finite-state machines. Journal of ACM 30, 323–342 (1983)
11. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2007), http://tata.gforge.inria.fr/
12. Eilenberg, S., Elgot, C.C., Shepherdson, J.C.: Sets recognized by n-tape automata. Journal of Algebra 13, 447–464 (1969)
13. Frougny, C., Sakarovitch, J.: Synchronized rational relations of finite and infinite words. Theoretical Computer Science 108, 45–82 (1993)
14. Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. Computing Research Repository (CoRR), abs/1011.0551 (2010)
15. Gold, E.M.: Language identication in the limit. Info. and Control 10(5), 447–474 (1967)
16. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
17. Harju, T., Karhumäki, J.: The equivalence problem of multitape finite automata. Theoretical Computer Science 78, 347–355 (1991)
18. Hill, J.L., Szewczyk, R., Woo, A., Hollar, S., Culler, D.E., Pister, K.S.J.: System architecture directions for networked sensors. In: Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 93–104 (2000)
19. Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M.F.: The Click modular router. ACM Transactions on Computer Systems 18(3), 263–297 (2000)
20. La Torre, S., Madhusudan, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 299–314. Springer, Heidelberg (2008)
21. Pachl, J.K.: Reachability problems for communicating finite state machines. Technical Report CS-82-12, Department of Computer Science, University of Waterloo (1982)
22. Pachl, J.K.: Protocol description and analysis based on a state transition model with channel expressions. In: Intl. Conf. on Protocol Specification, Testing and Verification (PSTV), pp. 207–219 (1987)
23. Pai, V.S., Druschel, P., Zwaenepoel, W.: Flash: An efficient and portable Web server. In: USENIX Annual Technical Conference, pp. 199–212 (1999)
24. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
25. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM Journal of Research and Development 3, 114–125 (1959)

26. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Transactions on Programming Languages and Systems 22, 416–430 (2000)
27. Raskin, J.-F., Servais, F.: Visibly pushdown transducers. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 386–397. Springer, Heidelberg (2008)
28. Sen, K., Viswanathan, M.: Model checking multithreaded programs with asynchronous atomic methods. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 300–314. Springer, Heidelberg (2006)
29. Thomas, W.: On logical definability of trace languages. In: ASMICS Workshop, Technical University of Munich, Report TUM-I9002, pp. 172–182 (1990)
30. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Learning to verify safety properties. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 274–289. Springer, Heidelberg (2004)

A Timed Component Algebra for Services

Benoît Delahaye[1], José Luiz Fiadeiro[2], Axel Legay[1], and Antónia Lopes[3]

[1] INRIA/IRISA, Rennes, France
{benoit.delahaye,axel.legay}@irisa.fr
[2] Dep. of Computer Science, Royal Holloway, University of London, UK
jose.fiadeiro@rhul.ac.uk
[3] Dep. of Informatics, Faculty of Sciences, University of Lisbon, Portugal
mal@di.fc.ul.pt

Abstract. We present a component algebra for services that can guarantee time-related properties. The components of this algebra are networks of processes that execute according to time constraints and communicate asynchronously through channels that can delay messages. We characterise a sub-class of consistent networks give sufficient conditions for that class to be closed under composition. Finally, we show how those conditions can be checked, at design time, over timed I/O automata as orchestrations of services, thus ensuring that, when binding a client with a supplier service at run time, the orchestrations of the two services can work together as interconnected without further checks.

1 Introduction

In [9,10], we revisited the notions of interface and component algebra proposed in [7] for component-based design and put forward elements of a corresponding interface theory for service-oriented design in which service orchestrations are networks of asynchronously communicating processes. That algebra is based on an implicit model of time: the behaviour of processes and channels is captured by infinite sequences of sets of actions, each action consisting of either the publication or the delivery of a message. However, such an implicit model of time is not realistic for modelling numerous examples of timed behaviour, from session timeouts to logical deadlines, and is not very effective for the analysis of properties. In this paper, we investigate an alternative model based on *timed traces* [3]. Even if this model assumes a minimal granularity of time, time is no longer implicit and, therefore, more realistic: we record the behaviour that is observed only at those instants of time when networks are active, not at every instant.

In this setting, we study the problem of ensuring consistency of run-time composition of orchestrations based on properties of processes and channels that can be checked at design time. This is important because run-time binding is an intrinsic feature of the service-oriented paradigm – one that distinguishes it from distributed systems in general – and that checking for consistency by actually calculating, at run time, the product of the automata that implement the services being bound to each other and checking for the non-emptiness of the resulting language is simply not realistic.

Not surprisingly, the results obtained in [10] for run-time composition under the implicit-time model do not extend directly to timed traces because the two spaces have different topological structures. Hence, one of our main contributions is the identification of refinement and closure operators that can support the composition of services that do not operate over the same time sequences.

The other main contribution results from adopting, as models of implementations of processes and channels, a variant of timed I/O automata (TIOA) that permits clock invariants on locations and in which all locations are Büchi accepting (as in [12]). Although results on the consistency of the composition of TIOA have been addressed in the literature they are based on a weaker notion of consistency according to which a TIOA that does not accept any non-Zeno timed sequence can still be consistent. In Sec. 5 we give an example of a situation in which the composition of two TIOA can only produce Zeno sequences, which is not acceptable as this would mean that joint behaviour would only be possible by forcing actions to be executed over successively shorter delays to converge on a deadline. The sub-algebra of TIOA that we characterise in Sec. 4 addresses this problem, i.e., we identify properties that can be checked, at design time, over networks of TIOA that ensure that, when binding a client with a supplier service, their orchestrations can operate together without further run-time checks.

2 The Component Algebra

We start by recalling a few concepts related to traces and their Cantor topology. Given a set A, a *trace* λ over A is an element of A^ω, i.e., an infinite sequence of elements of A. We denote by $\lambda(i)$ the $(i+1)$-th element of λ and by λ_i the prefix of λ that ends at $\lambda(i-1)$ if $i > 0$, with λ_0 being the empty sequence. A *segment* π is an element of A^*, i.e., a finite sequence of elements of A, the length of which we denote by $|\pi|$. We use $\pi{<}\lambda$ to mean that the segment π is a prefix of λ. Given $a{\in}A$, we denote by $(\pi{\cdot}a)$ the segment obtained by extending π with a. A *property* Λ over A is a set of traces. For every property Λ, we define $\Lambda^f = \{\pi : \exists\lambda{\in}\Lambda(\pi{<}\lambda)\}$ — the segments that are prefixes of traces in Λ, also called the *downward closure* of Λ — and $\bar{\Lambda} = \{\lambda : \forall\pi{<}\lambda(\pi{\in}\Lambda^f)\}$ — the traces whose prefixes are in Λ^f, also called the *closure* of Λ. A property Λ is said to be *closed* iff $\Lambda \supseteq \bar{\Lambda}$ (and, hence, $\Lambda = \bar{\Lambda}$).

In this timed model, every trace consists of an infinite sequence of pairs of an instant of time and a set of actions – the actions that are observed at that instant of time. In order to be able to model networks of systems, we allow that set of actions to be empty: on the one hand, this allows us to model finite behaviours, i.e., systems that stop executing actions after a certain point in time while still part of a network; on the other hand, it allows us to model observations that are triggered by actions performed by components outside the system.

This time model falls under what is often known as a 'point-based semantics', as opposed to an 'interval-based semantics' in which observations are made at every instant of time – our systems of systems are discrete and, therefore, a continuous observation model is not required. The advantages of the proposed model are that, on the one hand, it offers a natural extension of the trace-based model adopted in [10] and, on the other hand, it has been recently studied from the point of view of a number of decidability results [17].

Definition 1 (Timed traces). *Let A be a set (of actions) and $\delta \in \mathbb{R}_{>0}$.*

- *A* time sequence τ *is a trace over* $\mathbb{R}_{\geq 0}$ *for which there exists a sequence* $(d_i \in \mathbb{N}_+)_{i \in \mathbb{N}}$ *such that* $\tau(0) = 0$ *and* $\tau(i+1) = \tau(i) + d_i \times \delta$ *for every* i.
- *An* action sequence σ *is a trace over* 2^A *such that* $\sigma(0) = \emptyset$.
- *A* timed trace *over A is a pair $\lambda = \langle \sigma, \tau \rangle$ of an action and a time sequence. We denote by $\Lambda(A)$ the set of timed traces over A.*
- *Given a timed property $\Lambda \subseteq \Lambda(A)$, we define:*
  - *For every time sequence τ, $\Lambda_\tau = \{\sigma \in (2^A)^\omega : \langle \sigma, \tau \rangle \in \Lambda\}$ — the action property defined by Λ and τ.*
  - *$\Lambda_{time} = \{\tau : \exists \sigma \in (2^A)^\omega (\langle \sigma, \tau \rangle \in \Lambda)\}$ — the time sequences involved in Λ.*

The constant δ (fixed for the remainder of the paper) represents the minimal interval between two time observations — the sequence $(d_i)_{i \in \mathbb{N}}$ provides the duration associated with each step i. This implies in particular that time progresses, i.e., the set $\{\tau(i) : i \in \mathbb{N}\}$ is unbounded. Working with such a constant is realistic and endows the space of time sequences with topological properties that are stronger than those of the more general space of non-Zeno sequences.

Functions between sets of actions ('alphabet maps') are useful for defining relationships between processes and the networks in which they operate.

Definition 2 (Maps). *Let $f : A \to B$ be a function (alphabet map).*

- *For every $\sigma \in (2^B)^\omega$, we define $\sigma|_f \in (2^A)^\omega$ pointwise as $\sigma|_f(i) = f^{-1}(\sigma(i))$ — the* projection *of σ over A. If f is an inclusion, i.e., $A \subseteq B$, then we tend to write $\_|_A$ instead of $\_|_f$; this is a function that, when applied to a trace, forgets the actions of B that are not in A.*
- *For every timed trace $\lambda = \langle \sigma, \tau \rangle$ over B, we define its projection over A to be $\lambda|_f = \langle \sigma|_f, \tau \rangle$, and for every timed property Λ over B, $\Lambda|_f = \{\lambda|_f : \lambda \in \Lambda\}$ — the* projection *of Λ to A.*
- *For every timed property Λ over A, we define $f(\Lambda) = \{\langle \sigma, \tau \rangle : \langle \sigma|_f, \tau \rangle \in \Lambda\}$ — the* translation *of Λ to B.*

We are particularly interested in translations defined by prefixing every element of a set with a given symbol. Such translations are useful for identifying in a network the process to which an action belongs — we do not assume that processes have mutually disjoint alphabets. More precisely, given a set A and a symbol p, we denote by $(p.\_)$ the function that prefixes the elements of A with '$p.$'. Note that prefixing defines a bijection between A and its image $p.A$.

In our asynchronous model, interactions are based on the exchange of messages that are transmitted through channels. We organise messages in sets that we call ports: a *port* is a finite set (of messages). Ports are communication abstractions that are convenient for organising networks of processes.

Every message belonging to a port has an associated *polarity*: $^-$ if it is an outgoing message (published at the port) and $^+$ if it is incoming (delivered at the port). Therefore, every port M has a partition $M^- \cup M^+$. The actions of sending (publishing) or receiving (being delivered) a message m are denoted by $m!$ and $m¡$, respectively. More specifically, if M is a port, we define $A_{M^-}=\{m!{:}m\in M^-\}$, $A_{M^+}=\{m¡{:}m\in M^+\}$, and $A_M=A_{M^-}\cup A_{M^+}$ — the set of actions associated with M. Even if a process does not refuse the delivery of messages, it can discard them, e.g., if they arrive outside the protocol expected by the process, and a channel can accept the publication of every message but only deliver some published messages to their destination (this is for instance the case of unreliable channels).

A *process* consists of a finite set γ of mutually disjoint ports — i.e., each message that a process can exchange belongs to exactly one of its ports — and a non-empty timed property Λ over $A_\gamma = \bigcup_{M\in\gamma} A_M$ defining its behaviour.

Interactions are established through channels. A *channel* consists of a set M of messages and a non-empty timed property Λ over $A_M=\{m!, m¡ : m\in M\}$. Channels connect processes through their ports. Given ports M_1 and M_2 and a channel $\langle M, \Lambda\rangle$, a *connection* between M_1 and M_2 via $\langle M, \Lambda\rangle$ consists of a pair of injective maps $\mu_i{:}M\to M_i$ such that $\mu_i^{-1}(M_i^+) = \mu_j^{-1}(M_j^-)$, $\{i,j\}=\{1,2\}$ — i.e., a connection establishes a correspondence between the two ports such that any two messages that are connected have opposite polarities. Each μ_i is called the *attachment* of M to M_i. We denote the connection by the triple $\langle M_1 \overset{\mu_1}{\leftarrow} M \overset{\mu_2}{\rightarrow} M_2, \Lambda\rangle$. Notice that every connection defines an injection $\langle\mu_1,\mu_2\rangle$ from A_M to $A_{M_1}\cup A_{M_2}$ as follows: for every $m\in M$ and $\{i,j\}=\{1,2\}$, if $\mu_i(m)\in M_i^-$ then $\langle\mu_1,\mu_2\rangle(m!) = \mu_i(m)!$ and $\langle\mu_1,\mu_2\rangle(m¡) = \mu_j(m)¡$.

Definition 3 (t-ARN). *A timed asynchronous relational net consists of:*

- *A simple finite graph $\langle P, C\rangle$ where P is a set of nodes and C is a set of edges. Note that each edge is an unordered pair $\{p,q\}$ of nodes.*
- *A labelling function that assigns a process $\langle\gamma_p, \Lambda_p\rangle$ to every node p and a connection $\langle\gamma_c, \Lambda_c\rangle$ to every edge c such that:*
  - *If $c=\{p,q\}$ then γ_c is a pair of attachments $\langle M_p \overset{\mu_p}{\leftarrow} M_c \overset{\mu_q}{\rightarrow} M_q\rangle$ for some $M_p\in\gamma_p$ and $M_q\in\gamma_q$.*
  - *If $\gamma_{\{p,q\}}=\langle M_p \overset{\mu_p}{\leftarrow} M_{\{p,q\}} \overset{\mu_q}{\rightarrow} M_q\rangle$ and $\gamma_{\{p,q'\}}=\langle M'_p \overset{\mu'_p}{\leftarrow} M_{\{p,q'\}} \overset{\mu'_{q'}}{\longrightarrow} M'_{q'}\rangle$ with $q \neq q'$, then $M_p \neq M'_p$.*

*For every (*T*-ARN) α, we define the following sets and mappings:*

- *$A_\alpha = \bigcup_{p\in P} p.A_{\gamma_p}$ is the language associated with α.*
- *For every $p\in P$, ι_p is the function that maps A_{γ_p} to A_α, which prefixes the actions of A_{γ_p} with p.*
- *For every $c\in C$, ι_c is the function that maps A_{M_c} to A_α, which, assuming that $c = \{p,q\}$, translates the actions of A_{M_c} through $\langle p.\_\circ\mu_p, q.\_\circ\mu_q\rangle$.*
- *$\Lambda_\alpha = \{\lambda\in\Lambda(A_\alpha) : \forall p\in P(\lambda|_{\iota_p}\in\Lambda_p) \wedge \forall c\in C(\lambda|_{\iota_c}\in\Lambda_c)\}$.*

Note that, for every $p\in P$, $(\_|_{\iota_p})$ first removes the actions that are not in the language $p.A_p$ and then removes the prefix p. Similarly, for every $c=\{p,q\}\in C$,

$(\_|_{\iota_c})$ first removes the actions that are not in the language $\langle p.\_\circ\mu_p, q.\_\circ\mu_q\rangle(A_{M_c})$, then removes the prefixes p and q, and then projects onto the language of M_c.

As an example, consider a bank portal that mediates the interactions between clients and the bank in the context of different business operations such as a credit card request. Fig. 1 depicts a T-ARN with two interconnected processes that implement that business operation. Process *Clerk* is responsible for the interaction with the environment and for making decisions on credit card requests, for which it relies on the process *CreditValidator* that validates whether the requesters do not have bad credit (e.g., unpaid collections or recent offences). The behavior of these processes and the channel used for communication are subject to time-related constraints ensuring that the decision on a credit card request is always issued within twenty time units since the reception of the request.

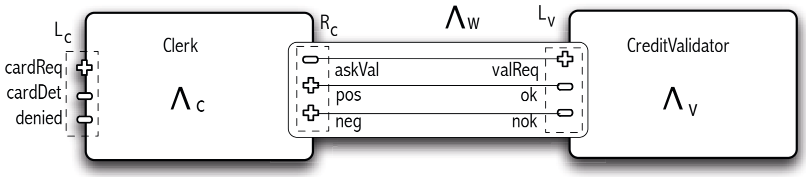


Fig. 1. An example of a T-ARN with two processes connected through a channel

The graph of the T-ARN consists of two nodes c:*Clerk* and v:*CreditValidator* and an edge $\{c, v\}$:w_{cv}.

- *Clerk* is a process with two ports. In port L_c, the process receives messages *cardReq* and sends *cardDet* (a message carrying the card details) and *denied*. Port R_c has outgoing message *askVal* and incoming messages *pos* and *neg*. The behaviour of *Clerk* is as follows. After the delivery of the first *cardReq* on port L_c, *Clerk* may either simply deny the card request by publishing *denied* or ask an external validation of the requester by publishing *askVal* on R_c. In both cases, the outgoing message is published within five time units since the reception of *cardReq*. Then, *Clerk* waits ten time units for the delivery of *pos* or *neg*, upon which it publishes within three time units, respectively, *cardDet* or *denied*. If none of these messages arrives by the deadline or both arrive together, *Clerk* publishes *denied* on L_c.
- *CreditValidator* is a process with a single port (L_v) with incoming message *valReq* and outgoing messages *ok* and *nok*. When the first *valReq* is delivered, it takes no more than seven time units to publish either *ok* or *nok*.
- The port R_c of *Clerk* is connected with the port L_e of *CreditValidator* through w_{cv}:$\langle R_c \overset{\mu_c}{\leftarrow} \{m, n, k\} \overset{\mu_v}{\rightarrow} L_v, \Lambda_w\rangle$, with μ_c=$\{m \mapsto askVal, n \mapsto pos, k \mapsto neg\}$, μ_e= $\{m \mapsto valReq, n \mapsto ok, k \mapsto nok\}$. The corresponding channel is reliable and introduces at most a delay of five time units in the transmission of messages: $msg¡$ follows within five time units the first $msg!$, for $msg \in \{m, n, k\}$.

We often refer to the T-ARN through the quadruple $\langle P, C, \gamma, \Lambda\rangle$ where γ returns the set of ports of the processes that label the nodes and the pair of attachments of the connections that label the edges, and Λ returns the corresponding properties. The fact that the graph is simple – undirected, without self-loops or multiple edges – means that all interactions between two given processes are supported by a single channel and that no process can interact with itself. The graph is undirected because, as already mentioned, channels are bidirectional. Furthermore, because of the second restriction on the labelling function, different channels cannot share ports.

The alphabet of A_α is the union of the alphabets of the processes involved translated by prefixing all actions with the node from which they originate (see the definition of this translation after Def. 2). We take the set Λ_α to define the set of possible traces observed on α – those traces over the alphabet of the T-ARN that are projected to traces of all its processes and channels. Notice that

$$\Lambda_\alpha = \bigcap_{p \in P} \iota_p(\Lambda_p) \cap \bigcap_{c \in C} \iota_c(\Lambda_c)$$

That is, the behaviour of the T-ARN is given by the intersection of the behaviour of the processes and channels translated to the language of the T-ARN — this corresponds to what one normally understands as a parallel composition. Notice that the translations applied to set of traces effectively open the behaviour of the processes and channels to actions in which they are not involved.

As in [9,10], two T-ARNs can be composed through the ports that are still available for establishing further interconnections, i.e., not connected to any other port, which we call interaction-points.

Definition 4 (Composition). *Let $\alpha_1 = \langle P_1, C_1, \gamma_1, \Lambda_1\rangle$ and $\alpha_2 = \langle P_2, C_2, \gamma_2, \Lambda_2\rangle$ be T-ARNs such that P_1 and P_2 are disjoint, and a family $w^i = \langle M_1^i \overset{\mu_1^i}{\leftarrow} M^i \overset{\mu_2^i}{\rightarrow} M_2^i, \Lambda^i\rangle$ $(i = 1 \ldots n)$ of connections for interaction-points $\langle p_1^i, M_1^i\rangle$ of α_1 and $\langle p_2^i, M_2^i\rangle$ of α_2 such that, for every $i \neq j$: (1) $p_1^i \neq p_1^j$ or $p_2^i \neq p_2^j$; (2) if $p_1^i = p_1^j$ then $M_1^i \neq M_1^j$; (3) if $p_2^i = p_2^j$ then $M_2^i \neq M_2^j$. The composition*

$$\alpha_1 \mathop{\Big|\Big|}\nolimits^{i=1\ldots n}_{\langle p_1^i, M_1^i\rangle, w^i, \langle p_2^i, M_2^i\rangle} \alpha_2$$

is the T-ARN defined as follows:

- *Its graph is $\langle P_1 \cup P_2, C_1 \cup C_2 \cup \bigcup_{i=1\ldots n}\{p_1^i, p_2^i\}\rangle$*
- *Its labelling function coincides with that of α_1 and α_2 on the corresponding subgraphs, and assigns to the new edges $\{p_1^i, p_2^i\}$ the label w^i.*

Fig. 1 also illustrates the composition of T-ARNs: the depicted T-ARN is the composition of the two atomic T-ARNs defined by *Clerk* and *CreditValidator*.

3 Consistency

In this section, we investigate conditions under which we can prove that a given T-ARN is consistent. Consistency is an important property of any component algebra [7]: in our setting, it establishes that the processes can work together as interconnected via the channels. We also aim for conditions that are closed under composition so that the consistency of a T-ARN can be derived from that of its parts. Our conditions rely on closure properties and a generalisation of the property of being 'progress-enabled' proposed in [10] for un-timed behaviour.

Definition 5 (Consistent t-ARN). *A* T-*ARN α is* consistent *if $\Lambda_\alpha \neq \emptyset$.*

In [10], we defined a sub-algebra of (un-timed) ARNs that are consistent and closed under composition. The characterisation of this sub-algebra relied on the closure operator induced by the Cantor topology over action sequences. The same closure operator can be defined over timed traces but, for the purpose of separating the properties required of the action sequences from those of the time sequences and the way they can be checked over automata (which we do in Sec. 4), it is useful to consider other notions of closure.

We can use the Cantor topology over $(2^A)^\omega$ to define a notion of closure relative to a fixed time sequence:

Definition 6 (Closure relative to time). *We say that a timed property Λ is* closed relative to time *or, simply, t-closed, iff, for every $\tau \in \Lambda_{time}$, Λ_τ is closed. A t-closed process/channel is one whose property is t-closed. A t-closed* T-*ARN is one in which all processes and channels are t-closed.*

Processes and channels that are closed relative to time define safety properties in the usual un-timed sense: over a fixed time sequence, which cannot be controlled by the processes or channels, the violation of the property can be checked over a finite trace. We consider now operations on time sequences.

Definition 7 (Time refinement). *Let $\rho{:}\mathbb{N}\to\mathbb{N}$ be a monotonically increasing function that satisfies $\rho(0)=0$.*

- *Let τ, τ' be two time sequences. We say that τ'* refines *τ through ρ, which we denote by $\tau' \preceq_\rho \tau$, iff, for every $i \in \mathbb{N}$, $\tau(i) = \tau'(\rho(i))$. We say that τ'* refines *τ, which we denote by $\tau' \preceq \tau$, iff $\tau' \preceq_\rho \tau$ for some ρ.*
- *Let $\lambda = \langle \sigma, \tau \rangle$, $\lambda' = \langle \sigma', \tau' \rangle$ be two timed traces. We say that λ' refines λ through ρ — which we denote by $\lambda' \preceq_\rho \lambda$ — iff $\tau' \preceq_\rho \tau$ and, for every $i \in \mathbb{N}$, $\sigma(i) = \sigma'(\rho(i))$ and, for every $\rho(i) < j < \rho(i+1)$, $\sigma'(j) = \emptyset$. We also say that λ' refines λ — which we denote by $\lambda' \preceq \lambda$ — iff $\lambda' \preceq_\rho \lambda$ for some ρ.*
- *The r-closure of a set Λ of timed traces is $\Lambda^r = \{\lambda' : \exists \lambda \in \Lambda (\lambda' \preceq \lambda)\}$*
- *We say that Λ is* closed under time refinement *or, simply, r-closed, iff $\Lambda^r \subseteq \Lambda$.*
- *An r-closed process/channel is one whose property is r-closed. An r-closed* T-*ARN is one in which all processes and channels are r-closed.*

That is, a time sequence refines another if the former interleaves time observations between any two time observations of the latter. Refinement extends to traces by requiring that no actions be observed in the finer trace between two consecutive times of the coarser trace. Therefore, the r-closure of a process adds all possible interleavings of empty observations to its traces, capturing its behaviour in any possible environment. This is related to mechanisms such as stuttering [1], which ensure that components do not constrain their environment.

It is not difficult to prove that the refinement relation is a complete meet semi-lattice, the meet of two time sequences τ_1 and τ_2 being given by the recursion

$$\tau(i+1) = min(\{\tau_1(j) > \tau(i), j \in \mathbb{N}\} \cup \{\tau_2(j) > \tau(i), j \in \mathbb{N}\})$$

together with the base $\tau(0) = 0$. It is also easy to prove that:

Proposition 8. *If a* T*-ARN α is t-closed (resp., r-closed), then Λ_α is also t-closed (resp., r-closed).*

A property that was found to be relevant in [10] for characterising consistent (un-timed) asynchronous relational nets concerns the ability to make joint progress. In the timed version, we analyse progress in relation to given time sequences.

Definition 9 (Progress-enabled). *For any* T*-ARN α and time sequence τ, let*

$$\Pi_{\alpha_\tau} = \{\pi \in (2^{A_\alpha})^*\colon \forall p \in P(\pi|_{\iota_p} \in \Lambda^f_{p_\tau}) \wedge \forall c \in C(\pi|_{\iota_c} \in \Lambda^f_{c_\tau})\}$$

We say that α is progress-enabled in relation to τ *iff*

$$\epsilon \in \Pi_{\alpha_\tau} \quad and \quad \forall \pi \in \Pi_{\alpha_\tau} (\exists \tau' \preceq \tau \exists A \subseteq A_\alpha ((\pi \cdot A) \in \Pi_{\alpha_{\tau'}} \wedge \tau'_{|\pi|} = \tau_{|\pi|}))$$

We say that α is progress-enabled *iff there is a time sequence τ such that α is progress-enabled in relation to every $\tau' \preceq \tau$.*

The set Π_{α_τ} consists of all the segments that the processes and channels can jointly engage in across the time sequence τ. Being progress-enabled relative to τ means that, after any initial joint segment, the processes and channels can make joint progress along a refinement of that time sequence. The reason for using a refinement of τ is that progress may depend on the activities performed at the interaction points of α. Note that, because the intersection of A with the alphabet of any process or channel can be empty, being progress-enabled does not require all parties to actually perform an action.

By itself, being progress-enabled does not guarantee that a T-ARN is consistent: moving from finite to infinite behaviours requires the analysis of what happens 'at the limit'. However, if we work with t-closed and r-closed properties, the limit behaviour will remain within the T-ARN:

Theorem 10. *A* T*-ARN is consistent if it is t-closed, r-closed and progress-enabled.*

We now show how T-ARNs can be guaranteed to be progress-enabled by construction. Every T-ARN that consists of a single node labelled with a process P is progress-enabled in relation to at least a time sequence. This is because processes are consistent. If we take the r-closure of P, then the T-ARN is progress-enabled. In [10], we gave criteria for the composition of two (un-timed) progress-enabled ARNs to be progress-enabled based on the ability of processes to buffer incoming messages – being 'delivery-enabled' – and of channels to buffer published messages – being 'publication-enabled'. In a timed domain, it becomes necessary to identify time sequences across which all parties can work together.

Definition 11 (Delivery-enabled). *Let $\alpha{=}\langle P, C, \gamma, \Lambda\rangle$, $\langle p, M\rangle{\in}I_\alpha$ one of its interaction-points, and $D_{\langle p,M\rangle}{=}\{p.m_{¡}\colon\ m{\in}M^+\}$. We say that α is delivery-enabled in relation to $\langle p, M\rangle$ if, for every $\tau{\in}\Lambda_{time}$, $(\pi{\cdot}A){\in}\Pi_{\alpha_\tau}$ and $B{\subseteq}D_{\langle p,M\rangle}$, there exists $\tau'{\preceq}\tau$ such that $(\pi{\cdot}B \cup (A{\setminus}D_{\langle p,M\rangle})){\in}\Pi_{\alpha_{\tau'}}$, and $\tau'_{|\pi|+1}{=}\ \tau_{|\pi|+1}$.*

That is, being delivery-enabled at an interaction point requires that, for every time sequence, any joint segment of the T-ARN over that sequence can be extended by any set of messages delivered at that interaction-point, after which it will behave according to a refinement of the original trace. Note that this does not interfere with the decision of the process to publish messages: $B{\cup}(A{\setminus}D_{\langle p,M\rangle})$ retains all the publications in A.

Definition 12 (Publication-enabled). *Let $h{=}\langle M{,}\Lambda\rangle$ be a channel and $E_h = \{m!{:}m{\in}M\}$. We say that h is publication-enabled iff, for every $\tau{\in}\Lambda_{time}$, $(\pi{\cdot}A){\in}\Lambda^f_\tau$ and $B{\subseteq}E_h$, there exists $\tau'{\preceq}\tau$ such that $\pi{\cdot}(B{\cup}(A{\setminus}E_h)){\in}\Lambda^f_{\tau'}$, and $\tau'_{|\pi|+1}{=}\ \tau_{|\pi|+1}$.*

The requirement here is that, for any time sequence and any segment of a trace over that time sequence, the segment can be extended by the publication of any set of messages, i.e., the channel should not prevent processes from publishing messages when they are in a state in which they could do so. Notice that this does not interfere with the decision of the channel to deliver messages: $(B{\cup}(A{\setminus}E_h))$ retains all the deliveries present in A.

Theorem 13. *Let α be a composition of r-closed progress-enabled T-ARNs through the connections $w^i = \langle M_1^i \overset{\mu_1^i}{\leftarrow} M^i \overset{\mu_2^i}{\rightarrow} M_2^i, \Lambda^i\rangle$, $i = 1 \ldots n$, i.e.,*

$$\alpha = (\alpha_1 \ \Big|\Big|^{i=1\ldots n}_{\langle p_1^i, M_1^i\rangle, w^i, \langle p_2^i, M_2^i\rangle} \ \alpha_2)$$

If, for $i{=}1\ldots n$, α_1 is delivery-enabled in relation to $\langle p_1^i{,}M_1^i\rangle$, α_2 is delivery-enabled in relation to $\langle p_2^i{,}M_2^i\rangle$ and $h^i{=}\langle M^i{,}\Lambda^i\rangle$ is publication-enabled and r-closed, then α is progress-enabled.

Therefore, the proof that an r-closed T-ARN is progress-enabled can be reduced to checking that individual processes are delivery-enabled in relation to their interaction points and that the channels used for composition are publication-enabled. To guarantee that the T-ARN is consistent, it is sufficient to choose processes and channels that are t-closed (implement safety properties). All the checking can be done at design time, not at composition time (which, in the service-oriented paradigm, is done at run time).

4 The Automata-Theoretic View

We now show how the properties introduced in the previous section can be checked over orchestrations of services based on automata-based models of processes and channels. We adopt Timed I/O Automata similar to those presented in [6], except that we use discrete time and sets of actions instead of single actions for transitions. As δ represents the minimal interval between two time observations, all the durations in the automata are in $\mathbb{N}_\delta^+$, the positive multiples of δ. We will use $\mathbb{N}_\delta$ to refer to $\mathbb{N}_\delta^+ \cup \{0\}$.

Let $\mathbb{C}$ be a finite set (of clocks). A *clock valuation* over $\mathbb{C}$ is a mapping v: $\mathbb{C} \to \mathbb{N}_\delta$. Given $d{\in}\mathbb{N}_\delta$ and a valuation v, we denote by $v{+}d$ the valuation defined by, for any clock $c{\in}\mathbb{C}$, $(v{+}d)(c) = v(c){+}d$. Given $R \subseteq \mathbb{C}$ and a clock valuation v, we denote by $v^{\mathbf{R}}$ the valuation where clocks from R are reset, i.e., such that $v^{\mathbf{R}}(c) = 0$ if $c{\in}R$ and $v^{\mathbf{R}}(c) = v(c)$ otherwise. Let *op* be the set of relational operators $op = \{\leq, \geq\}$. A *guard* over $\mathbb{C}$ is a finite conjunction of expressions of the form $c \bowtie n$ with $\bowtie{\in}op$ and $n{\in}\mathbb{N}$. We denote by $\mathcal{B}(\mathbb{C})$ the set of guards over $\mathbb{C}$.

Definition 14 (DTIOA). *A Discrete Timed I/O Automaton (DTIOA) is a tuple* $\mathcal{A} = \langle Loc, q_0, \mathbb{C}, E, Act, Inv\rangle$ *where:*

- *Loc is a finite set of locations and* $q_0{\in}Loc$ *is the initial location;*
- $\mathbb{C}$ *is a finite set of clocks;*
- $Act = Act^I \cup Act^O$ *is a finite set of actions partitioned into inputs and outputs;*
- $E \subseteq Loc \times 2^{Act} \times \mathcal{B}(\mathbb{C}) \times 2^{\mathbb{C}} \times Loc$ *is a finite set of edges;*
- Inv: $Loc \to \mathcal{B}(\mathbb{C})$ *associates an invariant with every location.*

In addition, we impose that every DTIOA is r-closed*: for all* $l{\in}Loc$, $(l, \emptyset, \phi, \emptyset, l){\in}E$ *for some valid* ϕ *in* $\mathcal{B}(\mathbb{C})$.

Being r-closed means that, in every location, it must be possible to make an empty observation without affecting the system. Intuitively, this reflects openness to environments that are involved in the execution of actions not included in *Act*.

An execution starting in location l_0 and clock valuation v_0 is an alternating sequence

$$(l_0, v_0, d_0) \overset{S_0,R_0}{\longrightarrow} (l_1, v_1, d_1) \overset{S_1,R_1}{\longrightarrow} \ldots$$

where: for every i, $l_i{\in}Loc$, v_i is a clock valuation over $\mathbb{C}$, $S_i{\subseteq}Act$ and $R_i{\subseteq}\mathbb{C}$; $d_0{\in}\mathbb{N}_\delta$ and, for $i{>}0$, $d_i{\in}\mathbb{N}_\delta^+$; and, for every i: (1) $Inv(l_i)(v_i + t)$ holds for all $0 \leq t \leq d_i$, (2) $v_{i+1}{=}(v_i + d_i)^{\mathbf{R_i}}$ and (3) there is $(l_i, S_i, C, R_i, l_{i+1}){\in}E$ such that $C(v_i + d_i)$ holds. The language of $\mathcal{A}$, which we denote by $\Lambda_{\mathcal{A}}$, is the set of executions such that $l_0{=}q_0$, $v_0(c) = 0$, for all $c{\in}\mathbb{C}$, and $d_0{>}0$.

In *deterministic* DTIOAs, for every location l and valuation v such that $Inv(l)(v)$ holds and $S \subseteq Act$, there exists at most one edge $(l, S, C, \_, \_){\in}E$ such that $C(v)$ holds. In this way, in these automata, the current state (l, v), the duration d of the stay in l and the next symbol S determine the next state (l', v') uniquely.

An execution in $\Lambda_{\mathcal{A}}$ defines a timed trace $\lambda{=}\langle\sigma, \tau\rangle$ over $Act^I{\cup}Act^O$ where $\sigma(0){=}\emptyset$, $\tau(0) = 0$ and, for $i \geq 0$, $\sigma(i + 1){=}S_i$ and $\tau(i + 1){=}\tau(i){+}d_i$. We denote by $[\![\mathcal{A}]\!]$ the set of timed traces defined by the set of executions in $\Lambda_{\mathcal{A}}$, which is r-closed in the sense of Def. 7. For example, the timed traces defined by the DTIOA in Fig. 2 are those in which either no input is ever received (in which case the system is idle forever) or, after the delivery of the first *valReq*, it takes no more than seven time units for the system to publish either *ok* or *nok* (after that, the system is open to inputs but does not publish anything more). Those traces correspond to the property that defines the behaviour of the process *CreditValidator* presented before.

A DTIOA $\mathcal{A}$ is *consistent* if $\Lambda_{\mathcal{A}}{\neq}\emptyset$ and has *consistent states* if, for every l and v such that $Inv(l)(v)$ holds, there exists an execution of $\mathcal{A}$ starting in (l, v). Notice that a DTIOA that has consistent states is not necessarily consistent. Indeed, although having consistent states implies that there is an infinite execution starting in the initial state, it could be the case that this execution has an initial duration $d_0{=}0$. Thus, in the following, we assume that DTIOA are consistent *and* have consistent states.

An important class of DTIOAs are those that are able to receive any set of inputs at all times (input-enabledness) and that, at each step, provide outputs that do not depend on the received inputs (independence). One way to ensure that DTIOA satisfy both requirements is to leverage the notions of delivery-enabledness and publication-enabledness of T-ARNs to the automata setting:

Definition 15 (DP-enabled DTIOA). *A DTIOA* $\mathcal{A} = \langle Loc, q_0, \mathbb{C}, E, Act, Inv\rangle$ *is* DP-enabled *if, for every* $B{\subseteq}Act^I$*, clock valuation* v*, and edge* $(l, A, C, R, l'){\in}E$ *such that the following properties hold —* $Inv(l)(v)$*,* $C(v)$ *and, for all* $0{\leq}t{\leq}\delta$*,* $Inv(l')(v^{\mathbf{R}}+t)$ *— there is an edge* $(l, B\cup(A\backslash Act^I), C', R', l''){\in}E$ *such that* $C'(v)$ *holds and, for all* $0 \leq t \leq \delta$*,* $Inv(l'')(v^{\mathbf{R}'} + t)$ *also holds.*

For a DTIOA to be input-enabled and independent, all edges need to be adaptable to accept any set of inputs without changing the associated outputs. Although the target locations of edges and clock resets may be modified when changing inputs, they are required to be enabled for execution at least in the same situations as the original ones.

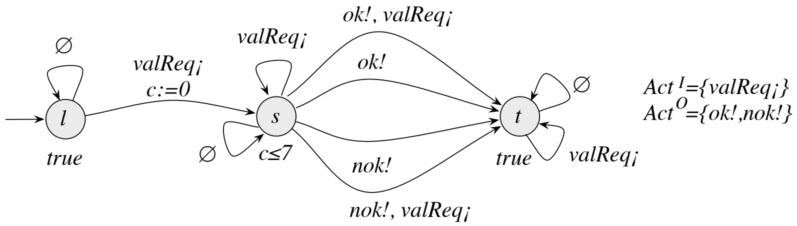


Fig. 2. An example of a DTIOA

Definition 16 (DTIOP). *A DTIO process consists of a set* $\gamma_{\mathcal{P}}{=}\{M_1,..., M_n\}$ *of mutually disjoint ports and a deterministic DP-enabled DTIOA* $\mathcal{A}_{\mathcal{P}}$ *that is consistent, has consistent states and for which* $Act^I{=}\cup_i A_{M_i^+}$ *and* $Act^O{=}\cup_i A_{M_i^-}$*.*

The inputs of a DTIOP are deliveries m¡ of incoming messages and outputs are publications $m!$ of outgoing messages at the ports. The language of a DTIOP is that of its DTIOA, i.e., $[\![\mathcal{P}]\!] = [\![\mathcal{A}_{\mathcal{P}}]\!]$. For example, the port L_V in Fig. 1 and the DTIOA in Fig. 2 define a DTIOP provided we choose $\delta<1$. The automaton is obviously deterministic, has consistent states and, if $\delta<1$, it is also DP-enabled.

As before, interconnection of DTIO processes is achieved through channel implementations, also defined in terms of DTIOA.

Definition 17 (DTIOC). *A DTIO channel (or DTIOC) consists of a set M of messages and a deterministic DP-enabled DTIOA $\mathcal{A}$ that is consistent, has consistent states and for which $Act^I=\{m!:m\in M\}$ and $Act^O=\{m¡:m\in M\}$.*

Notice that deliveries are outputs for channels (inputs for processes) and publications are inputs for channels (outputs for processes). This is because, messages published by a process are delivered to another process through a channel: if a process $\mathcal{P}_1$ is connected to a process $\mathcal{P}_2$ via a channel $\mathcal{C}$, the publication of a message m by $\mathcal{P}_1$ is an output for $\mathcal{P}_1$ and an input for $\mathcal{C}$; the delivery of m is an output for $\mathcal{C}$ and an input for $\mathcal{P}_2$.

Every DTIOP $\mathcal{P}$ defines a t-closed and r-closed process $P_{\mathcal{P}} = \langle \gamma_{\mathcal{P}}, [\![\mathcal{A}_{\mathcal{P}}]\!]\rangle$ in the sense of Sec. 2. Similarly, every DTIOC $\mathcal{C}=\langle M, \mathcal{A}\rangle$ defines a t-closed and r-closed channel $C_{\mathcal{C}}=\langle M, [\![\mathcal{A}_M]\!]\rangle$. Most importantly, $P_{\mathcal{P}}$ and $C_{\mathcal{C}}$ meet the conditions required for the application of Theo. 13.

Theorem 18. *If $\mathcal{P}$ is a DTIOP, then $P_{\mathcal{P}}$ is* DP-enabled *in relation to any of its ports and is* progress-enabled. *If $\mathcal{C}$ is a DTIOC, then $C_{\mathcal{C}}$ is* publication-enabled.

A DTIO net (or DTION) is defined in the same way as a T-ARN except that DTIOPs and DTIOCs are used instead of processes and channels, respectively. Every DTION $\mathcal{N}$ defines the T-ARN $\alpha_{\mathcal{N}}$ obtained by replacing the DTIOPs and DTIOCs with the corresponding processes and channels. By construction, $\alpha_{\mathcal{N}}$ is t-closed and r-closed. The semantics of a DTION can be defined in terms of the classical partially synchronized product of DTIOA, which we recall briefly.

Definition 19 (Product). *Two DTIOA $\mathcal{A}_i = \langle Loc^i, q_0^i, \mathbb{C}^i, E^i, Act_i, Inv^i\rangle$ are compatible iff $\mathbb{C}^1\cap\mathbb{C}^2=Act_1^I\cap Act_2^I=Act_1^O\cap Act_2^O=\emptyset$. The composition of two compatible DTIOA is $\mathcal{A}_1\|\mathcal{A}_2=\langle Loc^1 \times Loc^2, (q_0^1, q_0^2), \mathbb{C}^1 \cup \mathbb{C}^2, E, Act, Inv\rangle$ where:*

- $Act^I = (Act_1^I\backslash Act_2^O) \cup (Act_2^I\backslash Act_1^O)$
- $Act^O = Act_1^O\cup Act_2^O$
- *for all* $(q_1, q_2)\in Loc^1\times Loc^2$, $Inv((q_1, q_2))=Inv^1(q^1)\wedge Inv^2(q_2)$
- $((q_1, q_2), S, C, R, (q_1', q_2'))\in E$ *iff:* $(q_1, S_1, C_1, R_1, q_1')\in E^1$, $(q_2, S_2, C_2, R_2, q_2')\in E^2$, $C = C_1 \wedge C_2$, $S_i = S \cap Act_i$ *for* $i = 1, 2$, *and* $R = R_1 \cup R_2$.

Note that, by construction, when $S\cap Act_1 \neq \emptyset$ and $S\cap Act_2 \neq \emptyset$, all actions on which $\mathcal{A}_1$ and $\mathcal{A}_2$ synchronize (i.e., actions in $S\cap Act_1\cap Act_2$) are necessarily inputs on one side and outputs on the other. After composition these actions become outputs. Furthermore, transitions such that $S\cap Act_i = \emptyset$, which are usually considered as non-synchronizing, are handled as synchronizing transitions with underlying r-closure loops.

Proposition 20. *Given compatible DTIOA $\mathcal{A}_1$ and $\mathcal{A}_2$, $[\![\mathcal{A}_1 \| \mathcal{A}_2]\!] = \iota_1([\![\mathcal{A}_1]\!]) \cap \iota_2([\![\mathcal{A}_2]\!])$, where ι_1 and ι_2 translate the local languages to that of the composition, as in Def. 3.*

In order to show that this notion of product can be used to capture the semantics of DTIONs and that this semantics is compositional, consider the simple case of a DTION $\mathcal{N}$ consisting of two nodes p_1 and p_2 labelled with $\mathcal{P}_1 = \langle \gamma_1, \mathcal{A}_{\mathcal{P}_1} \rangle$ and $\mathcal{P}_2 = \langle \gamma_2, \mathcal{A}_{\mathcal{P}_2} \rangle$, respectively, and an edge c between them labelled with the connection $\mathcal{C} = \langle M_1 \overset{\mu_1}{\leftarrow} M \overset{\mu_2}{\rightarrow} M_2, \mathcal{A}_M \rangle$ where $\langle M, \mathcal{A}_M \rangle$ is a DTIOC. We use prefixing as in Sec. 2, i.e., we denote by $p.\mathcal{A}$ the copy of $\mathcal{A}$ where all actions are prefixed by p.

The connection $\mathcal{C}$ defines a DTIOA $\mathcal{A}_\mathcal{C}$ that is a copy of $\mathcal{A}_M$ except that the alphabet is renamed using the injection $\langle p_1.\_\circ \mu_1, p_2.\_\circ \mu_2 \rangle$ defined as in Sec. 2 to enforce synchronization of $\mathcal{P}_1$ and $\mathcal{P}_2$ on the ports M_1 and M_2: given a message $m{\in}M$, the action $m¡$ is renamed $p_i.m¡$ if $\mu_i(m){\in}M_i^+$ and the action $m!$ is renamed $p_i.m!$ if $\mu_i(m){\in}M_i^-$. More precisely, if $\mathcal{A}_M = \langle Loc^M, q_0^M, \mathbb{C}^M, E^M, Act^M, Inv^M \rangle$, then $\mathcal{A}_\mathcal{C} = \langle Loc^M, q_0^M, \mathbb{C}^M, E^C, Act_C, Inv^M \rangle$ where:

- $Act_C^O = \{p_1.m¡ : m \in M_1^+ \cap \mu_1(M)\} \cup \{p_2.m¡ : m \in M_2^+ \cap \mu_2(M)\}$
- $Act_C^I = \{p_1.m! : m \in M_1^- \cap \mu_1(M)\} \cup \{p_2.m! : m \in M_2^- \cap \mu_2(M)\}$
- $E^C = \{(q_c, \langle p_1.\_ \circ \mu_1, p_2.\_ \circ \mu_2 \rangle(A), C, R, q_c') : (q_c, A, C, R, q_c') \in E^M\}$

We can now define the semantics of $\mathcal{N}$ as the product of $p_1.\mathcal{A}_{\mathcal{P}_1}$, $p_2.\mathcal{A}_{\mathcal{P}_2}$ and $\mathcal{A}_\mathcal{C}$. Notice that, because of the renamings, the three DTIOA are compatible and $\mathcal{A}_\mathcal{C}$ ensures that the synchronization only occurs between messages that are related through the maps μ_1 and μ_2. In other words, given a message $m{\in}M$ such that $\mu_1(m) = m_1{\in}M_1^-$ and $\mu_2(m) = m_2{\in}M_2^+$: the action $m!$ from $\mathcal{A}_M$ is renamed as $p_1.m_1!$ in $\mathcal{A}_\mathcal{C}$ (an input), and will thus synchronize with $m_1!$ of $\mathcal{P}_1$ (an output), i.e., $p_1.\mu_1(m)!$ in the composition — $\mathcal{P}_1$ synchronizes with $\mathcal{C}$ to publish m_1; the action $m¡$ from $\mathcal{A}_M$ is renamed as $p_2.m_2¡$ in $\mathcal{A}_\mathcal{C}$ (an output), and will thus synchronize with $m_2¡$ of $\mathcal{P}_2$ (an input), i.e., $p_2.\mu_2(m)$ in the composition — $\mathcal{C}$ synchronizes with $\mathcal{P}_2$ to deliver the message. Because of the renaming of the actions of $\mathcal{P}_1$ and $\mathcal{P}_2$ by prefixing them with p_1 and p_2, respectively, no other synchronizations take place.

Theorem 21 (Compositional semantics). *Given a DTION $\mathcal{N}$ as above,*

$$[\![\mathcal{N}]\!] = [\![p_1.\mathcal{A}_{\mathcal{P}_1} \parallel \mathcal{A}_\mathcal{C} \parallel p_2.\mathcal{A}_{\mathcal{P}_2}]\!] = \iota_{p_1}([\![\mathcal{P}_1]\!]) \cap \iota_c([\![\mathcal{C}]\!]) \cap \iota_{p_2}([\![\mathcal{P}_2]\!]) = \Lambda_{\alpha_\mathcal{N}}$$

That is, the semantics of $\mathcal{N}$ is $\alpha_\mathcal{N}$: the product of the DTIOAs that implement the processes and connections of the net generates the set of timed traces obtained through Def. 3 – the semantics of the corresponding T-ARN. The result can be generalized to arbitrary DTIONs by calculating the products corresponding to all interconnections.

5 Related Work

Several frameworks have been proposed for component/service-based software systems that exhibit timed properties. Some, such as [15,16], adopt the π-calculus to address subclasses of timing activities, e.g., timeouts and local urgency in web transactions. Others adopt an algebraic framework: for example, [4] adopts timed data streams for a channel-based coordination model, and [8,11,14,18] address service choreography using timed automata, i.e., they focus on the modelling of the (timed) conversation protocols that characterise the global behaviour of a (fixed) number of peers that exchange services. One of the properties that the latter analyse is compatibility – whether the conversation protocols (modelled as timed automata) followed by the peers lead to deadlocks or time conflicts that prevent them from completing (e.g., reaching final states).

Although compatibility relates to the notion of consistency that we address in this paper, our emphasis is not on choreography but on orchestration: what we are investigating is in what conditions we can guarantee that the orchestrations of two services can work together when they bind to each other. This has implications on the properties that are required of timed-automata in order to guarantee consistency. Because we aim to support run-time binding and composition, those properties are different from those investigated for choreography (where composition is analysed at design time). An example is the way time is managed: in choreography, this is done globally for the (fixed) set of peers (in the sense that clocks can be set or reset by all peers); in our approach, this needs to be done locally at level of each process because composition is dynamic.

The interaction model is another key aspect of a theory of services. Most service models are synchronous even if message-passing is more adequate for the loosely-coupled operating environment of services [14]. An asynchronous timed model is considered in [11], but only indirectly by simulating buffers in a synchronous setting, which limits the properties that can be analysed.

The problem of guaranteeing the consistency of composition without having to calculate the product of automata (or other models of orchestration) has remained largely ignored in the literature (e.g., [5,6]) as its relevance comes to the fore in service-oriented computing thanks to the crucial distinction that needs to be made between design-time and run-time checks or operations.

Results on the consistency of the composition of Timed I/O Automata (TIOA) are addressed in [6] for the restricted class of TIOA that are input-enabled and allow independent progress, which are directly relevant for our paper. However, their results are based on a weaker notion of consistency according to which a TIOA that does not accept any non-Zeno timed sequence can still be consistent, which is not sufficient to ensure that the composition of TIOA accepting non-empty sets of timed traces is a TIOA that also accepts at least one timed trace. Fig. 3 illustrates this situation: both automata allow independent progress, are input-enabled, and can produce infinite timed traces; however, their composition yields a TIOA that can only produce Zeno sequences.

The same class of TIOA is considered in [13], where their I/O feasibility is investigated. Although this is richer than consistency because in this context TIOA executions are also not necessarily time divergent. Hence, the results that establish sufficient conditions for the composition of I/O feasible TIOA to be

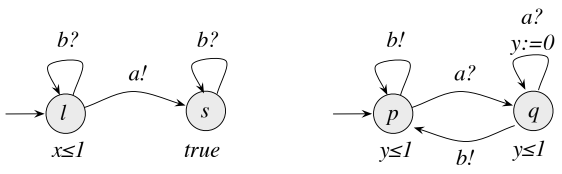


Fig. 3. Input and progress-enabled TIOA that do not generate any joint trace

I/O feasible (based on progressive and receptive TIOA) cannot be transposed to the world of sets of timed traces.

6 Concluding Remarks

In this paper, we have investigated how a component algebra can be defined over timed traces that addresses run-time composition of services. Services are orchestrated by asynchronous networks of processes and can bind dynamically to required services. Our results include the characterisation of a sub-algebra over which the binding can be proved to be consistent using only design-time properties of the orchestrations, i.e., without having to make further checks at run time (which would undermine real-time operation). We showed how discrete timed I/O automata provide a compositional implementation model for that algebra and identified a class of DTIOA that conform to the properties that ensure consistency of composition – those that are deterministic and DP-enabled. These results extend the literature on TIOA, which so far had not addressed the issues raised by run-time composition.

Our model uses a time unit (in the domain of the reals) for observations but it is not discrete in the sense that, because clock valuations are not restricted to the time unit, the behaviour of TIOA can be constrained by real-time guards. However, this time granularity is shared by all processes. Although this is adequate for service-level agreements in typical business transactions, a non-discrete model would allow us to capture heterogeneity and address a more general class of systems. However, non-Zeno models fail to satisfy the topological properties over which we rely to ensure consistency of networks, namely that refinement defines a complete meet semi-lattice, which could lead to situations in which joint behaviour is only possible by forcing actions to be executed over successively shorter delays to converge on a deadline. We are currently investigating intermediate models over more restricted structures of actions.

We are also investigating t-closure over DTIOA in relation to traditional characterisations of safety properties over time sequences, e.g., nondeterministic Büchi automata [2]. This will allow us to use logics such as Safety MTL [17] to define an interface algebra for T-ARNs similar to [9,10] and investigate the use of model-checking techniques for validating orchestrations in relation to interfaces.

Acknowledgments. This work was partially supported by Fundação para a Ciência e Tecnologia under contract (PTDC/EIA-EIA/103103/2008).

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. 82(2), 253–284 (1991)
2. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Computing 2(3), 117–126 (1987)
3. Alur, R., Henzinger, T.A.: Logics and models of real time: A survey. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 74–106. Springer, Heidelberg (1992)
4. Arbab, F., Rutten, J.J.M.M.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2003. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003)
5. Chilton, C., Kwiatkowska, M.Z., Wang, X.: Revisiting timed specification theories: A linear-time perspective. In: Jurdziński, M., Ničković, D. (eds.) FORMATS 2012. LNCS, vol. 7595, pp. 75–90. Springer, Heidelberg (2012)
6. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Timed I/O automata: a complete specification theory for real-time systems. In: HSCC, pp. 91–100. ACM (2010)
7. de Alfaro, L., Henzinger, T.A.: Interface theories for component-based design. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 148–165. Springer, Heidelberg (2001)
8. Díaz, G., Pardo, J.J., Cambronero, M.-E., Valero, V., Cuartero, F.: Verification of web services with timed automata. Electr. Notes Theor. Comput. Sci. 157(2), 19–34 (2006)
9. Fiadeiro, J.L., Lopes, A.: An interface theory for service-oriented design. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 18–33. Springer, Heidelberg (2011)
10. Fiadeiro, J.L., Lopes, A.: Consistency of service composition. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 63–77. Springer, Heidelberg (2012)
11. Guermouche, N., Godart, C.: Timed model checking based approach for web services analysis. In: ICWS, pp. 213–221. IEEE (2009)
12. Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Inf. Comput. 111(2), 193–244 (1994)
13. Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: The Theory of Timed I/O Automata. Morgan & Claypool Publishers (2006)
14. Kazhamiakin, R., Pandya, P.K., Pistore, M.: Representation, verification, and computation of timed properties in web. In: ICWS, pp. 497–504. IEEE Computer Society (2006)
15. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 282–298. Springer, Heidelberg (2005)
16. Lapadula, A., Pugliese, R., Tiezzi, F.: C-clock-WS: A timed service-oriented calculus. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 275–290. Springer, Heidelberg (2007)
17. Ouaknine, J., Worrell, J.: Safety metric temporal logic is fully decidable. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 411–425. Springer, Heidelberg (2006)
18. Ponge, J., Benatallah, B., Casati, F., Toumani, F.: Analysis and applications of timed service protocols. ACM Trans. Softw. Eng. Methodol. 19(4) (2010)

Probabilistic Analysis of the Quality Calculus

Hanne Riis Nielson and Flemming Nielson

DTU Compute, Technical University of Denmark, Denmark
{hrni,fnie}@dtu.dk

Abstract. We consider a fragment of the Quality Calculus, previously introduced for defensive programming of software components such that it becomes natural to plan for default behaviour in case the ideal behaviour fails due to unreliable communication.

This paper develops a probabilistically based *trust* analysis supporting the Quality Calculus. It uses information about the probabilities that expected input will be absent in order to determine the trustworthiness of the data used for controlling the distributed system; the main challenge is to take accord of the stochastic dependency between some of the inputs. This takes the form of a relational static analysis dealing with quantitative information.

1 Introduction

Motivation. Distributed systems often need to continue operating in a meaningful way even when communication links become unreliable. This is particularly important in embedded systems, control systems and systems providing critical infrastructure services. Such systems form key components of cyber physical systems where the consequences of software malfunction may be disastrous. As an example, if the engine control system of a car breaks down when the car is driving at high speed, a crash might result.

Unreliability of communication links due to security attacks is relatively well understood. There are numerous cryptographic protocols for ensuring the confidentiality, integrity and authenticity of data communicated between components of distributed systems. Also there are many state-of-the-art tools for finding and eliminating security flaws in such protocols. As an example, forcing a car to brake because the tyre sensors incorrectly report loss of pressure, can be avoided using proper cryptographic protocols (although it would add to the cost of the pressure sensors and transmitters).

Unreliability of communication links due to faults and denial of service attacks is much harder to guard against. In a cyber physical system it seems hard to avoid that the communication link can be flooded with irrelevant messages, that there might be radio interference on the wireless frequency band, or that physical antennas are shielded from receiving or transmitting their signals. This calls for utilising programming notations that help to ensure that software systems are hardened against such attacks on communication. As an example, the braking system of a car should be designed such that some braking effect continues to be applied even if no signals are received from the braking sensors about the spinning (and potential blocking) of the wheels.

We consider here the Quality Calculus [7], that is a recent proposal for how to enforce robustness considerations on software components executing in an open and error prone environment: what should the behaviour be when communication links may have broken down. While this is a first step in this important direction it is necessary to develop analyses to indicate the trust that we can have in the overall robustness of the system.

In this paper we develop a novel analysis for supporting the Quality Calculus. It is a probabilistically based *trust* analysis that uses information about the probabilities that expected input will be absent in order to determine the trustworthiness of the data used for controlling the distributed system; the main challenge is to take accord of the stochastic dependency between some of the probabilities determined in the availability analysis.

Overview. A fragment of the Quality Calculus [7] is reviewed in Section 2. Its distinguishing feature is a *binder* specifying the inputs to be performed before continuing. In the simplest case it is an input guard $t^{\ell}?x$ describing that some value should be received over the channel t and should be bound to the variable x; data over t is assumed to have trustworthiness as given by an element ℓ from some finite lattice of trust values. Increasing in complexity there are binders of the form $\&_q(t_1^{\ell_1}?x_1, \cdots, t_n^{\ell_n}?x_n)$ indicating that several inputs are simultaneously active and with a quality predicate q that determines when sufficient inputs have been received to continue. As a consequence, when continuing with the process after the binder, some variables might not have obtained proper values as the corresponding inputs have not been performed. To model this the calculus distinguishes between data and optional data, much like the use of option data types in programming languages like Standard ML. The construct $\mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2$ will evaluate the variable x; if it evaluates to $\mathsf{some}(c)$ we will execute P_1 with y bound to c; if it evaluates to none we will execute P_2.

The main motivating example, an intelligent smart meter, is discussed in Section 3. It is a scenario inspired by [8] where a smart meter of a household communicates with a service provider to obtain a schedule for operating a number of appliances taking pricing and availability of energy into account.

In Section 4 we show how to make use of information about the probabilities that expected input will be absent. We develop a probabilistic trust analysis associating probability distributions with all program points of interest where the probabilities indicate the trust level of the optional data. This helps in pinpointing those parts of the code where there is a high risk of basing decisions on less trustworthy data (like default data). The main challenge is to adequately model when the probabilities are stochastically dependent and when they are not; in the parlance of static analysis this means that we need to perform relational rather than independent attribute analyses of the Quality Calculus [6]

Table 1. A fragment of the Quality Calculus

$$\begin{aligned} P &::= (\nu c^{\ell})\, P \;\mid\; P_1 \,|\, P_2 \;\mid\; \mathsf{0} \;\mid\; b.P \;\mid\; t_1!t_2.P \;\mid\; A \\ &\;\;\mid\; \mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y) \colon P_1\ \mathsf{else}\ P_2 \\ b &::= \&_q(t_1^{\ell_1}?x_1, \cdots, t_n^{\ell_n}?x_n) \\ t &::= y \;\mid\; c \;\mid\; g(t_1, \cdots, t_n) \end{aligned}$$

and our development constitutes an extension of relational analyses to deal with quantitative information. We conclude and present our outlook on future work in Section 5.

2 Review of the Quality Calculus

A main purpose of process calculi is to focus on programming abstractions that may provide insight into the development of distributed systems. The challenge of concurrency was addressed in early process calculi like CCS [4] and the π-calculus [5] whereas the challenge of Service Oriented Computation have been addressed in calculi such as COWS [3], SOCK [2]and CaSPiS [1]. We consider here a fragment of the Quality Calculus [7] that addresses the challenge of how to ensure robustness of software systems that execute in an open environment, that does not always live up to expectations — possibly because anticipated communications do not take place due to faults or denial of service attacks.

Syntax. We now develop the syntax and semantics of a fragment of the Quality Calculus by adapting the development of [7]. A *system* consists a number of process definitions and a main process:

$$\mathsf{define}\ A_1 \triangleq P_1; \cdots; A_n \triangleq P_n\ \mathsf{in}\ P_*\ \mathsf{using}\ c_1^{\ell_1}, \cdots, c_m^{\ell_m}$$

Here A_i is the name of a process, P_i is its body, P_* is the main process and $c_1, \cdots, c_m$ is a list of constants. The syntax of processes is given in Table 1. A *process* can have the form $(\nu c^{\ell})\, P$ introducing a new constant c and its scope P, it can be a parallel composition $P_1 \,|\, P_2$ of two processes P_1 and P_2 and it can be an empty process denoted $\mathsf{0}$. An input process is written $b.P$ where b is a binder (explained below) specifying the inputs to be performed before continuing with P. An output process has the form $t_1!t_2.P$ specifying that the value t_2 should be communicated over the channel t_1. A process can also be a call A to one of the defined processes or a case construct (explained below). We shall feel free to dispense with trailing occurrences of the process $\mathsf{0}$.

To indicate the trust level of data we use an element ℓ from a finite trust lattice $\mathcal{L}$ and we write $\leq$ for the ordering on $\mathcal{L}$. As an example, we might use $\mathcal{L} = (\{\textsc{l}, \textsc{m}, \textsc{h}\}, \leq)$ to denote that available data is classified into low (L), medium (M) or high (H) trust and with the obvious linear ordering on these elements. More refined examples may be obtained by adopting the lattices of Mandatory

Access Control Policies or the Decentralised Label Model. All constants are annotated with an element from the trust lattice when they are introduced into the system (either in the list $c_1^{\ell_1}, \cdots, c_m^{\ell_m}$ or using the syntax $(\nu c^{\ell})\, P$); these annotations are performed by the programmers as part of a code review for determining the quality of the code.

The main distinguishing feature of the Quality Calculus is the *binder* b specifying the inputs to be performed before continuing. In the simplest case it is an input guard $t^{\ell}?x$ describing that some value should be received over the channel t and it will be bound to the variable x; the trust element ℓ indicates the amount of trust to be placed in data received over this channel. Increasing in complexity we may have binders of the form $\&_q(t_1^{\ell_1}?x_1, \cdots, t_n^{\ell_n}?x_n)$ indicating that n inputs are simultaneously active and a *quality predicate* q determines when sufficient inputs have been received to continue.

As an example, q can be $\exists$ meaning that one input is required, or it can be $\forall$ meaning that all inputs are required; formally $\exists(x_1, \cdots, x_n) \Leftrightarrow x_1 \vee \cdots \vee x_n$ and $\forall(x_1, \cdots, x_n) \Leftrightarrow x_1 \wedge \cdots \wedge x_n$. For more expressiveness we shall allow to write e.g. $[0 \wedge (1 \vee 2)]$ for the quality predicate defined by $[0 \wedge (1 \vee 2)](x_0, x_1, x_2) \Leftrightarrow x_0 \wedge (x_1 \vee x_2)$; this is particularly useful because unlike [7] we do not allow to nest binders.

As a consequence, when continuing with the process P in $b.P$ some variables might not have obtained proper values as the corresponding inputs might not have been performed. To model this we distinguish between *data* and *optional data*, much like the use of option data types in programming languages like Standard ML. In the syntax we use constants c, functions g (taking data as arguments and producing data as results), variables y and terms t to denote data and we use variables x to denote optional data; in particular, the expression $\mathsf{some}(t)$ signals the presence of some data t and none the absence of data. Returning to the processes, the construct $\mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y) : P_1\ \mathsf{else}\ P_2$ will test whether x evaluates to some data and if so, bind it to y and continue with P_1 and otherwise continue with P_2.

We need to impose a few well-formedness constraints on systems. Names u are divided into data constants c, data variables y, and optional data variables x. For a system of the form displayed above we shall require that the main process P_* as well as the bodies P_i have no free variables (over data or over optional data) and that their free constants are among $c_1, \cdots, c_m$.

Semantics. The semantics consists of a structural congruence and a transition relation [5]. The *structural congruence* $P_1 \equiv P_2$ is defined in Table 2 and expresses when two processes, P_1 and P_2, are congruent to each other. It enforces that processes constitute a monoid with respect to parallel composition and the empty process and it takes care of the unfolding of calls of named processes and scopes for constants; here $\mathsf{fc}(P)$ is the set of constants occurring free in P. Finally, it allows replacement in contexts C given by:

$$C ::= [\,] \mid (\nu c^{\ell})\, C \mid C|P \mid P|C$$

Table 2. Structural congruence of the Quality Calculus

$P \equiv P$	$P_1 \equiv P_2 \Rightarrow P_2 \equiv P_1$
$P_1 \equiv P_2 \wedge P_2 \equiv P_3 \Rightarrow P_1 \equiv P_3$	$P \,\vert\, \mathsf{0} \equiv P$
$P_1 \,\vert\, P_2 \equiv P_2 \,\vert\, P_1$	$P_1 \,\vert\, (P_2 \,\vert\, P_3) \equiv (P_1 \,\vert\, P_2) \,\vert\, P_3$
$(\nu c^\ell)\, P \equiv P \quad \text{if } c \notin \mathsf{fc}(P)$	$(\nu c_1^{\ell_1})\,(\nu c_2^{\ell_2})\, P \equiv (\nu c_2^{\ell_2})\,(\nu c_1^{\ell_1})\, P \quad \text{if } c_1 \neq c_2$
$A \equiv P \quad \text{if } A \triangleq P$	$(\nu c^\ell)\,(P_1 \mid P_2) \equiv ((\nu c^\ell)\, P_1) \mid P_2 \quad \text{if } c \notin \mathsf{fc}(P_2)$
$P_1 \equiv P_2 \Rightarrow C[P_1] \equiv C[P_2]$	

The *transition relation*

$$P \longrightarrow P'$$

describes when a process P evaluates into another process P'. It is parameterised on a relation $t \rhd c$ describing when a (closed) term t evaluates to a constant c; the definition of this relation is straightforward and hence omitted. Furthermore, we shall make use of two auxiliary relations

$$c_1!c_2 \vdash b \to b'$$

for specifying the effect on the binder b of matching the output $c_1!c_2$, and

$$b ::_v \theta$$

for recording (in $v \in \{\mathsf{tt}, \mathsf{ff}\}$) whether or not all required inputs of b have been performed as well as information about the composite substitution (θ) that has been constructed. To formalise this we extend the syntax of binders to include substitutions as well as individual inputs

$$b ::= \&_q(b'_1, \cdots, b'_n) \qquad \text{with} \qquad b' ::= [\mathsf{some}(c)/x] \;\mid t^\ell?x$$

where $[\mathsf{some}(c)/x]$ is the substitution that maps x to $\mathsf{some}(c)$ and leaves all other variables unchanged. We shall write *id* for the identity substitution and $\theta_2\theta_1$ for the composition of two substitutions, so $(\theta_2\theta_1)(x) = \theta_2(\theta_1(x))$ for all x.

The first part of Table 3 defines the transition relation $P \longrightarrow P'$. The first clause expresses that the original binder is replaced by a new binder recording the output just performed; this transition is only possible when $b ::_{\mathsf{ff}} \theta$ holds, meaning that more inputs are required before proceeding with the continuation P_2. The second clause considers the case where no further inputs are required; this is expressed by the premise $b ::_{\mathsf{tt}} \theta$. In this case the binding is performed by applying the substitution θ to the continuation process — hence no further inputs are allowed. The next clauses are straightforward; they define the semantics of the case construct, how the structural congruence is embedded in the transition relation and how transitions take place in contexts.

The next clause in Table 3 defines the auxiliary relation $c_1!c_2 \vdash b \to b'$; here the idea is simply to record the binding of the value received in the appropriate position.

Table 3. Transition rules of the Quality Calculus

$$\frac{t_1 \rhd c_1 \quad t_2 \rhd c_2 \quad c_1!c_2 \vdash b \to b' \quad b' ::_{\mathsf{ff}} \theta}{t_1!t_2.P_1 \mid b.P_2 \longrightarrow P_1 \mid b'.P_2}$$

$$\frac{t_1 \rhd c_1 \quad t_2 \rhd c_2 \quad c_1!c_2 \vdash b \to b' \quad b' ::_{\mathsf{tt}} \theta}{t_1!t_2.P_1 \mid b.P_2 \longrightarrow P_1 \mid P_2\theta}$$

$$\mathsf{case\ some}(c)\ \mathsf{of\ some}(y)\colon P_1\ \mathsf{else}\ P_2 \longrightarrow P_1[c/y]$$

$$\mathsf{case\ none\ of\ some}(y)\colon P_1\ \mathsf{else}\ P_2 \longrightarrow P_2$$

$$\frac{P_1 \equiv P_2 \quad P_2 \longrightarrow P_3 \quad P_3 \equiv P_4}{P_1 \longrightarrow P_4} \qquad \frac{P_1 \longrightarrow P_2}{C[P_1] \longrightarrow C[P_2]}$$

$$\frac{t \rhd c_1}{c_1!c_2 \vdash \&_q(b_1, \cdots, t^{\ell}?x, \cdots, b_n) \to \&_q(b_1, \cdots [\mathsf{some}(c_2)/x], \cdots, b_n)}$$

$$t^{\ell}?x ::_{\mathsf{ff}} [\mathsf{none}/x] \qquad [\mathsf{some}(c)/x] ::_{\mathsf{tt}} [\mathsf{some}(c)/x]$$

$$\frac{b_1 ::_{v_1} \theta_1 \quad \cdots \quad b_n ::_{v_n} \theta_n}{\&_q(b_1, \cdots, b_n) ::_v \theta_n \cdots \theta_1} \text{ where } v = q(v_1, \cdots, v_n)$$

The auxiliary relation $b ::_v \theta$ is defined in the final group of clauses in Table 3. Here we perform a pass over the (extended) syntax of the binder b, evaluating whether or not a sufficient number of inputs have been performed and recorded in v, and computing the associated composite substitution θ.

3 Motivating Example

We now introduce our main motivating example for the analyses to be developed subsequently. It is a scenario inspired by [8] where a smart meter SM of a household communicates with a service provider SP to obtain a schedule for operating a number of appliances taking pricing and availability of energy into account. Unfortunately, the service provider is subject to denial of service attacks and therefore the smart meter is equipped with a local computer LC for computing a schedule to be used whenever the service provider does not produce a schedule. The schedule computed by the service provider is the preferred one so the smart meter will use a clock CL to set a waiting time; only if no schedule is received

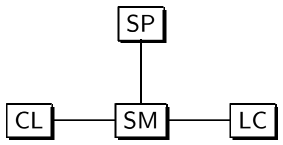


Fig. 1. The Smart Meter scenario

$$\begin{array}{ll}
\mathsf{define} & \mathsf{SP} \triangleq \cdots \text{ (see text) } \cdots \\
 & \mathsf{LC} \triangleq \cdots \text{ (see text) } \cdots \\
 & \mathsf{CL} \triangleq \cdots \text{ (see text) } \cdots \\
 & \mathsf{SM} \triangleq \cdots \text{ (see text) } \cdots \\
\mathsf{in} & \mathsf{SP} \mid \mathsf{LC} \mid \mathsf{Clock} \mid \mathsf{SM} \\
\mathsf{using} & \mathsf{request}^{\mathrm{H}}, \mathsf{req}^{\mathrm{H}}, \mathsf{rep}^{\mathrm{L}}, \mathsf{set}^{\mathrm{H}}, \mathsf{tick}^{\mathrm{H}}, \mathsf{install}^{\mathrm{H}}, \checkmark^{\mathrm{H}}
\end{array}$$

Fig. 2. The Smart Meter system

from the service provider within the waiting time, the smart meter will use the locally computed schedule. The overall scenario is illustrated in Figures 1 and 2 and we shall now describe the individual processes in more detail; we shall feel free to use a polyadic version of the calculus when it eases the presentation and also to use basic actions delay and wait for indicating that time passes (as explained below) but otherwise having no effect in the semantics.

The service provider SP is defined by:

$$\begin{array}{l}
\mathsf{SP} \triangleq \&_{\exists}(\mathsf{request}^{\mathrm{H}}?(x_i x_r)).\ \mathsf{case}\ x_i\ \mathsf{of}\ \mathsf{some}(y_i):\\
\qquad\qquad\qquad\qquad \mathsf{case}\ x_r\ \mathsf{of}\ \mathsf{some}(y_r):\ \mathsf{delay}_g.y_i!(\mathsf{global}(y_r)).\mathsf{SP}\\
\qquad\qquad\qquad\qquad \mathsf{else\ SP}\\
\qquad\qquad\qquad \mathsf{else\ SP}
\end{array}$$

The service provider first obtains a request of x_r resources from a smart meter identifying itself as x_i; the annotation H indicates that the channel request carries information of high trust level. The next three lines express that SP after some delay computes a schedule using the function global (taking data as input and returning data) and sends it to the smart meter before recursing. The case constructs ensure that the proper data is extracted and supplied to the function; indeed the two else branches are not reachable.

The local computer LC is similar to the service provider except that it makes use of the channels req and rep for obtaining the request and returning the reply:

$$\mathsf{LC} \triangleq \&_{\exists}(\mathsf{req}^{\mathrm{H}}?x_r).\ \mathsf{case}\ x_r\ \mathsf{of}\ \mathsf{some}(y_r):\ \mathsf{delay}_l.\mathsf{rep}!\mathsf{local}(y_r).\mathsf{LC}\ \mathsf{else}\ \mathsf{LC}$$

It uses the function local (taking data as input and returning data) to compute the schedule; once more the case construct is used to extract the actual request and the else branch is in fact not reachable.

As already mentioned the smart meter will put a limit on how long time it will wait for a schedule and the process CL below models a simple clock communicating over the channels set and tick:

$$\mathsf{CL} \triangleq \&_{\exists}(\mathsf{set}^{\mathrm{H}}?x_t).\ \mathsf{case}\ x_t\ \mathsf{of}\ \mathsf{some}(y_t):\ \mathsf{wait}(c).\mathsf{tick}!\checkmark.\mathsf{CL}\ \mathsf{else}\ \mathsf{CL}$$

The idea is that the input on set starts the clock and the output of the constant ✓ on the channel tick indicates that the prescribed waiting time c has passed.

Finally, let us consider the main control process SM for the smart meter:

$$\begin{array}{ll}\mathsf{SM} \triangleq (\nu i_g^{\mathrm{M}})\,(\nu d^{\mathrm{H}}) & \mathsf{request!}(i_g d).\ \mathsf{req!}d.\ \mathsf{set!}\checkmark. \\ & \&_{[0\wedge(1\vee 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l). \\ & \mathsf{case}\ x_g\ \mathsf{of\ some}(y_g)\colon {}^{1}\mathsf{install!}y_g.\mathsf{SM} \\ & \mathsf{else\ case}\ x_l\ \mathsf{of\ some}(y_l)\colon {}^{2}\mathsf{install!}y_l.\mathsf{SM} \\ & \quad\quad \mathsf{else}\ {}^{3}\mathsf{0}\end{array}$$

For later reference we have added labels to three subprocesses. The first line issues two requests for a schedule, one to the service provider and one to the local computer and it also starts the clock. The binder of the second line expresses that the time must have passed and that at least one schedule must have been received before continuing. Note that it is indeed possible for both schedules to arrive before the time has passed and it is also possible that no schedule has arrived when the time has passed in which case the smart meter continues waiting. The third line will give priority to the global schedule (the process labelled 1) whereas in the absence of a global schedule the fourth line will install the local schedule before recursing (the process labelled 2). As in the previous examples the case constructs are used to extract the required data and in fact the final else branch (labelled 3) is not reachable.

Discussion. As an alternative we might use the binder

$$\&_{[0\vee(1\wedge 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$$

in line 2. Then we will stop waiting for schedules when the time bound has been met but in the case where both schedules have been received it is possible to continue before the specified time has passed. Changing the binder to

$$\&_{[0\vee(1\vee 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$$

means that we proceed as soon as one of the schedules has arrived or the time has passed; as we shall see later this is the least attractive of the three models as concerns the robustness of the smart meter.

4 Trust Analysis

The Quality Calculus allows to describe which data is needed before continuing but given the expressiveness of the quality predicates one cannot be sure what data has actually been received. The subsequent process is able to determine this by testing for whether given data has actually been received and decisions can be made accordingly. Some of these decisions may have high trustworthiness whereas others may have low trustworthiness. In order to evaluate the overall system it is therefore important

– to be able to *annotate* the binders with information about the probability distribution over the trustworthiness of data actually received, and
– to be able to *propagate* these trust annotations throughout the process.

Trust Annotations. To annotate the binders we change the syntax of binders from $\&_q(t_1^{\ell_1}?x_1, \cdots, t_n^{\ell_n}?x_n)$ to $\&_q^{\pi}(t_1^{\ell_1}?x_1, \cdots, t_n^{\ell_n}?x_n)$. Here

$$\pi \in \mathcal{D}(\{x_1, \cdots, x_n\} \to \mathcal{L}_\perp)$$

is a probability distribution indicating the probability of the various inputs having been received where $\perp$ denotes the absence of input and $\mathcal{L}_\perp$ is the lifted trust lattice obtained from $\mathcal{L}$ by adding $\perp$ as the new least element. Note that the distribution assigns probabilities to tuples of inputs rather than to individual inputs because the quality predicates may be such that the various inputs are not stochastically independent — this is where our development will constitute a generalisation of relational static analyses to take care of quantitative information (in this case the probabilities).

One way to obtain the probability distribution is to run a given process for a while and to sample the actual distribution of inputs at the binders. Another way is to assume that the delays are governed by memoryless processes started just before the binders and then to use stochastic techniques to estimate the distributions. We shall illustrate the latter approach through a few examples.

Example 1. Let us consider the process SM of Section 3 and let us assume that the processes computing the global and local schedules are exponentially distributed with rates λ_g and λ_l, respectively. We want to determine the distribution π that should be used in the following binder:

$$\&^{\pi}_{[0\wedge(1\vee 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$$

Let us assume that $\lambda_g = 0.2$ and $\lambda_l = 0.5$ and that the waiting time (c) is 5 time units. One can show that

$$\begin{aligned}
\pi([x_t \mapsto \mathrm{H},\ x_g \mapsto \mathrm{M},\ x_l \mapsto \mathrm{L}]) &= 0.580\\
\pi([x_t \mapsto \mathrm{H},\ x_g \mapsto \mathrm{M},\ x_l \mapsto \perp] &= 0.061\\
\pi([x_t \mapsto \mathrm{H},\ x_g \mapsto \perp,\ x_l \mapsto \mathrm{L}] &= 0.359
\end{aligned}$$

and $\pi(\sigma) = 0$ in all other cases.

Example 2. Let us next consider the variant of SM using the binder

$$\&^{\pi}_{[0\vee(1\wedge 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$$

and let us, as before, assume that the processes computing the global and local schedules are exponentially distributed with rates λ_g and λ_l, respectively. Taking $\lambda_g = 0.2$ and $\lambda_l = 0.5$ we get the following distribution when the waiting time is 5 time units:

$$\begin{aligned}
\pi([x_t \mapsto \perp,\ x_g \mapsto \mathrm{M},\ x_l \mapsto \mathrm{L}]) &= 0.580\\
\pi([x_t \mapsto \mathrm{H},\ x_g \mapsto \mathrm{M},\ x_l \mapsto \perp]) &= 0.052\\
\pi([x_t \mapsto \mathrm{H},\ x_g \mapsto \perp,\ x_l \mapsto \mathrm{L}]) &= 0.338\\
\pi([x_t \mapsto \mathrm{H},\ x_g \mapsto \perp,\ x_l \mapsto \perp]) &= 0.030
\end{aligned}$$

and $\pi(\sigma) = 0$ in all other cases. As an example $\pi([x_t \mapsto \mathrm{H}, x_g \mapsto \mathrm{M}, x_l \mapsto \mathrm{L}]) = 0$ because it would correspond to the event that one of x_g and x_l is received before

Table 4. Trust Analysis of the Quality Calculus

$$\vdash 1, \pi_* \,@\, P_* \qquad \vdash 1, \pi_* \,@\, P_1 \qquad \cdots \qquad \vdash 1, \pi_* \,@\, P_n$$

$$\frac{\vdash p, \pi \,@\, (\nu c^\ell)\, P}{\vdash p, (\pi|^{\mathsf{C}}_{\{c\}}) \otimes \delta_{[c \mapsto \ell]} \,@\, P} \qquad \frac{\vdash p, \pi \,@\, (P_1 \mid P_2)}{\vdash p, \pi \,@\, P_1} \qquad \frac{\vdash p, \pi \,@\, (P_1 \mid P_2)}{\vdash p, \pi \,@\, P_2}$$

$$\frac{\vdash p, \pi \,@\, (b.P) \qquad \vdash b \blacktriangleright \pi_b}{\vdash p, (\pi|^{\mathsf{C}}_{\mathsf{bv}(b)}) \otimes \pi_b \,@\, P} \qquad \frac{\vdash p, \pi \,@\, (t_1!t_2.P)}{\vdash p, \pi \,@\, P}$$

$$\frac{\vdash p, \pi \,@\, (\mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y) \colon P_1\ \mathsf{else}\ P_2)}{\vdash p \cdot \pi_{[x \neq \bot]}, \pi_{\downarrow[x \neq \bot]}[y := x] \,@\, P_1} \quad \text{if } \pi_{[x \neq \bot]} \neq 0$$

$$\frac{\vdash p, \pi \,@\, (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{some}(y) \colon P_1\ \mathsf{else}\ P_2)}{\vdash p \cdot \pi_{[x = \bot]}, \pi_{\downarrow[x = \bot]} \,@\, P_2} \quad \text{if } \pi_{[x = \bot]} \neq 0$$

time has passed and the other is received at the exact moment that time passes; while this is a possible event it occurs with probability 0.

Example 3. Changing the binder to

$$\&^{\pi}_{[0 \vee (1 \vee 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$$

gives rise to the following distribution using the same parameters as above:

$$\begin{aligned} \pi([x_t \mapsto \bot,\ x_g \mapsto \mathrm{M},\ x_l \mapsto \bot]) &= 0.277 \\ \pi([x_t \mapsto \bot,\ x_g \mapsto \bot,\ x_l \mapsto \mathrm{L}]) &= 0.693 \\ \pi([x_t \mapsto \mathrm{H},\ x_g \mapsto \bot,\ x_l \mapsto \bot]) &= 0.030 \end{aligned}$$

and $\pi(\sigma) = 0$ in all other cases.

We shall find it helpful to use the notation $\vdash \&^{\pi}_{q}(t_1^{\ell_1}?x_1, \cdots, t_n^{\ell_n}?x_n) \blacktriangleright \pi$ in order to extract the probability annotation from the binder.

Trust Propagation. We now consider how to propagate the trust information from the binders throughout the entire system. The judgements of our analysis of processes P take the form

$$\vdash p, \pi \,@\, P$$

Here p is the probability that we will reach this occurrence of the process P and π is a distribution from $\mathcal{D}(V \to \mathcal{L}_\bot)$ where the free names of P are contained in the set V (determined by the detailed definition of the analysis). The mappings of $V \to \mathcal{L}_\bot$ assign trust levels to the free names of P and the idea is that π specifies the distribution of these mappings when P is reached. The mappings $\sigma : V \to \mathcal{L}_\bot$ will ensure that constants c and variables over data y only take values in $\mathcal{L}$.

A mapping $\sigma : V \to \mathcal{L}_\perp$ gives rise to distribution δ_σ in $\mathcal{D}(V \to \mathcal{L}_\perp)$, called the *Dirac distribution*, by assigning it the probability 1 and all other mappings the probability 0:

$$\delta_\sigma(\sigma') = \begin{cases} 1 \text{ if } \sigma = \sigma' \\ 0 \text{ otherwise} \end{cases}$$

We shall need a number of operations on $\mathcal{D}(V \to \mathcal{L}_\perp)$. The first is a *projection* on a subset $U \subseteq V$ of names, written $\pi\,|_U$. Given a distribution π in $\mathcal{D}(V \to \mathcal{L}_\perp)$ it will return a distribution $\pi\,|_U$ in $\mathcal{D}(U \to \mathcal{L}_\perp)$ and is defined by

$$(\pi\,|_U)(\sigma) = \Sigma_{(\sigma' \text{ s.t. } \sigma=\sigma'|_U)}\pi(\sigma')$$

Here $\sigma'\,|_U$ is the restriction of the mapping $\sigma' : V \to \mathcal{L}_\perp$ to the domain U and we thus summarise all the probabilities associated with mappings that are equal when projected on U. We shall write $\pi|^{\mathsf{C}}_U$ as a shorthand for $\pi\,|_{V\setminus U}$, that is for projection on the complement of U:

$$(\pi|^{\mathsf{C}}_U)(\sigma) = \Sigma_{(\sigma' \text{ s.t. } \sigma=\sigma'|_{V\setminus U})}\pi(\sigma')$$

In Table 4 we shall use this operator when binding new constants in $(\nu c^\ell)\,P$ and new variables in $b.P$ — the point being that old instances of these name will no longer be in scope.

The next operation is a *product* operation; it takes two distributions $\pi_1 : \mathcal{D}(V_1 \to \mathcal{L}_\perp)$ and $\pi_2 : \mathcal{D}(V_2 \to \mathcal{L}_\perp)$ defined over disjunct sets of names (so $V_1 \cap V_2 = \emptyset$) as arguments and constructs a distribution $\pi_1 \otimes \pi_2$ in $\mathcal{D}((V_1 \cup V_2) \to \mathcal{L}_\perp)$. It is given by

$$(\pi_1 \otimes \pi_2)(\sigma) = \pi_1(\sigma\,|_{V_1}) \cdot \pi_2(\sigma\,|_{V_2})$$

Thus we split the mapping into two parts and multiply the probabilities obtained from the two distributions. We shall use this operation when combining two stochastically independent distributions. For the construct $(\nu c^\ell)\,P$ we take the product with the Dirac distribution $\delta_{[c\mapsto\ell]}$ and in the case of the input binder $b.P$ we take the product with the distribution π_b obtained from the binder b using the notation $\vdash \&^\pi_q(t_1^{\ell_1}?x_1, \cdots, t_n^{\ell_n}?x_n) \blacktriangleright \pi$.

The next operation is a simple *lookup* operation: given a distribution $\pi : \mathcal{D}(V \to \mathcal{L}_\perp)$, a name u and a trust level ℓ we shall be interested in the probability $\pi_{[u=\ell]}$ for u having the trust level ℓ:

$$\pi_{[u=\ell]} = \Sigma_{(\sigma \text{ s.t. } \sigma(u)=\ell)}\pi(\sigma)$$

In a similar way we can define $\pi_{[u\neq\ell]}$ as the probability that u does not have the trust level ℓ:

$$\pi_{[u\neq\ell]} = \Sigma_{(\sigma \text{ s.t. } \sigma(u)\neq\ell)}\pi(\sigma)$$

It is easy to see that $\pi_{[u\neq\ell]} = 1 - \pi_{[u=\ell]}$. These operations are used in the analysis of the construct $\mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y)\colon P_1\ \mathsf{else}\ P_2$; here $\pi_{[x\neq\perp]}$ is the probability that the first branch is taken and similarly $\pi_{[x=\perp]}$ is the probability that the second branch is taken.

We shall also introduce a *selection* operation that given a distribution $\pi : \mathcal{D}(V \to \mathcal{L}_\perp)$, a name u and a trust level ℓ will construct a new distribution $\pi_{\downarrow[u \neq \ell]}$ of $\mathcal{D}(V \to \mathcal{L}_\perp)$ that gives 0 for all mappings σ with $\sigma(u) = \ell$ and rescales the remaining probabilities. The operation is only defined when $\pi_{[u \neq \ell]} \neq 0$:

$$(\pi_{\downarrow[u \neq \ell]})(\sigma) = \begin{cases} \frac{\pi(\sigma)}{\pi_{[u \neq \ell]}} & \text{if } \sigma(u) \neq \ell \\ 0 & \text{if } \sigma(u) = \ell \end{cases}$$

In a similar way we can define the operation $\pi_{\downarrow[u = \ell]}$ that gives 0 for all mappings σ with $\sigma(u) \neq \ell$ and rescales the remaining probabilities. This operation is only defined when $\pi_{[u = \ell]} \neq 0$:

$$(\pi_{\downarrow[u = \ell]})(\sigma) = \begin{cases} 0 & \text{if } \sigma(u) \neq \ell \\ \frac{\pi(\sigma)}{\pi_{[u = \ell]}} & \text{if } \sigma(u) = \ell \end{cases}$$

These operations are used for the construct $\mathsf{case}\ x\ \mathsf{of}\ \mathsf{some}(y) \colon P_1\ \mathsf{else}\ P_2$. Here the distribution for the first branch will be $\pi_{\downarrow[x \neq \perp]}$ assuming that it might be taken, that is, that $\pi_{[x \neq \perp]} \neq 0$. The distribution for the second branch is $\pi_{\downarrow[x = \perp]}$, again under the assumption that it might be taken, that is, that $\pi_{[x = \perp]} \neq 0$.

Finally, we need an operation for the *extension* of a distribution $\pi : \mathcal{D}(V \to \mathcal{L}_\perp)$ with a new name $u' \notin V$ that is stochastically dependent on a name $u \in V$ in the sense that u and u' have the same trust level. The resulting distribution $\pi[u' := u]$ is in $\mathcal{D}((V \cup \{u'\}) \to \mathcal{L}_\perp)$ and it is defined by

$$(\pi[u' := u])\sigma = \begin{cases} \pi(\sigma|^{\complement}_{\{u'\}}) & \text{if } \sigma(u) = \sigma(u') \\ 0 & \text{otherwise} \end{cases}$$

This operation is used in the analysis of the case construct where the distribution for the first branch needs to be extended with information about the variable y being bound by the construct so the distribution will be $(\pi_{\downarrow[x \neq \perp]})[y := x]$.

We now have the machinery needed for the definition of $\vdash p, \pi \,@\, P$ in Table 4. We analyse the main process and all bodies with the initial choice of $p = 1$ and $\pi = \pi_*$ where the list of constants $c_1^{\ell_1}, \cdots, c_m^{\ell_m}$ in the system definition in Section 2 gives rise to defining $\pi_* = \delta_{[c_1 \mapsto \ell_1, \cdots, c_m \mapsto \ell_m]}$. The choice of $p = 1$ reflects that all probabilities of reaching program points are conditional on the main process being called as well as the bodies. The remaining clauses were explained when introducing the auxiliary notation.

The analysis has been implemented using Standard ML. For the examples shown here it suffices to represent distributions π as lists of pairs (σ, p) where σ is a mapping and p is a non-zero probability. A number of improvements are possible. We may use the fact that variables in the domain of distributions are often stochastically independent (as when introduced in different binders) and hence distributions could be represented as products of distributions over disjoint domains. Also we may use symbolic techniques (like Binary Decision Diagrams) to represent the distributions that cannot be split into the product of distributions over simpler domains.

Querying the Analysis. We shall give two examples of how information about outputs can be extracted from the analysis.

First let us consider an output of the form $c!t$ and assume we are interested in the trust levels of the values being communicated over the channel c. Furthermore let us assume that the analysis gives us a probability p and a distribution π satisfying $\vdash p, \pi \,@\, c!t.P$. This means that the program point of interest is reached with probability p and furthermore, the distribution will be π. The trust levels of the values being communicated over c can be represented as a distribution ξ_π in $\mathcal{D}(\mathcal{L})$ and it can be defined by:

$$\xi_\pi(\ell) = \Sigma_{(\sigma \text{ s.t. } \sigma(t)=\ell)} \pi(\sigma)$$

Here $\sigma(t)$ is the trust level of the term t; for constants and variable this is straightforward as the information is available in σ whereas for terms $g(t_1, \cdots, t_n)$ we need to specify how to compute the resulting trust level from the trust levels of the arguments. So we shall assume that we have interpretations

$$\langle\!\langle g \rangle\!\rangle : \mathcal{L} \times \cdots \times \mathcal{L} \to \mathcal{L}$$

specifying this; as an example we might take $\langle\!\langle g \rangle\!\rangle(\ell_1, \cdots, \ell_n) = \ell_1 \sqcap \cdots \sqcap \ell_n$ for $\sqcap$ being the greatest lower bound of the trust lattice $\mathcal{L}$ and reflecting that a result is no more trustworthy than any of the arguments used to compute it.

Next let us consider all outputs of the form $c!\cdot$ and let us make two simplifying assumptions. The first is that the constant c is always explicit (i.e. does not arise in the form of some $y!\cdot$ where y is eventually replaced by c). The other is that all occurrences of $c!\cdot$ occur in different branches of the case constructs and that in particular no occurrence of $c!\cdot$ prefixes another. We may then calculate the distribution Ξ_c in $\mathcal{D}(\mathcal{L}_\perp)$ of the trust levels of data communicated over c. For a trust level $\ell \in \mathcal{L}$ we define

$$\Xi_c(\ell) = \Sigma_{\vdash p, \pi \,@\, c!t.P} \big(\Sigma_{\sigma \text{ s.t. } \sigma(t)=\ell} p \cdot \pi(\sigma) \big)$$

where the first sum is over all occurrences of $\vdash p, \pi \,@\, c!t.P$ in the analysis of the overall system of interest. For the trust level $\perp$ we define

$$\Xi_c(\perp) = 1 - \Sigma_{\ell \in \mathcal{L}}\, \Xi_c(\ell)$$

so as to reflect the probability that no communication over c is taking place.

Analysing the Motivating Example. We have used our implementation to analyse the smart meter example in various scenarios.

Let us first consider Example 1 where the binder used in the definition of SM is $\&^{\pi}_{[0\wedge(1\vee 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$ and where the processes computing the schedules are exponentially distributed with rates $\lambda_g = 0.2$ and $\lambda_l = 0.5$. This corresponds to a scenario where the service provider is somewhat slower to deliver a schedule than the local computer.

Using the trust analysis we can then compute the probability Ξ_{install} of the smart meter installing a schedule computed by the service provider, a locally computed schedule or no schedule at all; this corresponds to combining the analysis results for the program points labelled 1, 2, and 3 in the process SM in Section 3. The result is $\Xi_{\mathsf{install}}(\mathrm{H}) = 0$, $\Xi_{\mathsf{install}}(\mathrm{M}) = 0.64$, $\Xi_{\mathsf{install}}(\mathrm{L}) = 0.36$, and $\Xi_{\mathsf{install}}(\bot) = 0$. This shows that we are always sure to get a schedule installed and in 64% of the cases it is the global schedule and in the remaining 36% it is the local schedule. The information about $\Xi_{\mathsf{install}}(\mathrm{H})$, $\Xi_{\mathsf{install}}(\mathrm{M})$, and $\Xi_{\mathsf{install}}(\mathrm{L})$ originate from the points labelled 1 and 2 in the code for SM in Section 3, whereas the information about $\Xi_{\mathsf{install}}(\bot)$ originates from the point labelled 3.

Let us next consider Example 2 where the binder used in the definition of SM is $\&^{\pi}_{[0\vee(1\wedge 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$ and let us repeat the experiments using the same parameters as above. The result is $\Xi_{\mathsf{install}}(\mathrm{H}) = 0$, $\Xi_{\mathsf{install}}(\mathrm{M}) = 0.63$, $\Xi_{\mathsf{install}}(\mathrm{L}) = 0.34$, and $\Xi_{\mathsf{install}}(\bot) = 0.03$. This shows that we can no longer be sure to get a schedule installed but that the global schedule is still much more likely than the local schedule.

Finally, let us consider Example 3 where the binder used in the definition of SM is $\&^{\pi}_{[0\vee(1\vee 2)]}(\mathsf{tick}^{\mathrm{H}}?x_t, i_g^{\mathrm{M}}?x_g, \mathsf{rep}^{\mathrm{L}}?x_l)$ and let us repeat the experiments using the same parameters as above. The result is $\Xi_{\mathsf{install}}(\mathrm{H}) = 0$, $\Xi_{\mathsf{install}}(\mathrm{M}) = 0.28$, $\Xi_{\mathsf{install}}(\mathrm{L}) = 0.69$, and $\Xi_{\mathsf{install}}(\bot) = 0.03$. This shows that the risk of getting no schedule has not changed but that it is much more likely to be the local schedule than the global schedule.

5 Conclusion

Software is increasingly being used in malign rather than benign environments and this calls for adopting a more defensive programming style. Existing security errors are to a large extent due to an over-optimistic programming style where programmers focus on giving the software as much functionality as possible. The remedy is to adopt a more defensive programming style where programmers focus on avoiding errors that can be caused by external components — in particular, due to communication becoming unreliable.

There are numerous examples of software systems being developed in one context and then ported to another. In this process the threat scenario might change and changes to the software may be called for. As an example, the destruction of the Ariane 5 rocket on its maiden voyage was due to the reuse of software from Ariane 4 that suddenly operated outside of the previous design parameters, thereby causing a floating point overflow disrupting the safe control of the rocket.

We believe that suitable programming notations form a key ingredient in motivating programmers to produce more robust code. We also believe that such programming notations can be studied in a pure form, in the context of process calculi, as done in the present paper. In this paper the focus has solely been on the consequences of disruption to the communication links. This paves the way for dialects of main stream programming languages enforcing such robustness considerations and taking into account also the need to use cryptographic communication protocols to ensure confidentiality, integrity and authenticity of communication.

The main technical contribution of this paper is the development of a probabilistic trust analysis able to identify the extent to which the robust programming style has been adhered to.

The probabilistically based trust analysis indicates the places where decisions are made based on data of limited trustworthiness. It amounts to propagating the probability distributions from the binders throughout the program. This is done in such a way that the stochastic dependence between input variables is preserved and properly conditioned when passing through case statements. A naive implementation of probability distributions may lead to state explosion; more efficient methods may utilise symbolic data structures (like Binary Decision Diagrams) and sparse array techniques similar to those used in model checkers for Discrete Time Markov Chains.

In our view, this analysis forms the basis for supporting a new discipline of robust programming. We believe that it will be able to address the risks posed by porting software from one environment to another; by recalculating the probabilities one can better determine whether or not the software continues to deal appropriately with risks and threats in the new application environment.

Acknowledgement. The research has been supported by MT-LAB, a VKR Centre of Excellence for the Modelling of Information Technology, and by IDEA4CPS, funded by the Danish Foundation for Basic Research under grant DNRF86-10.

References

1. Bruni, R.: Calculi for service-oriented computing. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 1–41. Springer, Heidelberg (2009)
2. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., 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)
3. 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)
4. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
5. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press (1999)
6. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis (2. corr. print). Springer (2005)
7. Nielson, H.R., Nielson, F., Vigo, R.: A Calculus for Quality. In: Păsăreanu, C.S., Salaün, G. (eds.) FACS 2012. LNCS, vol. 7684, pp. 188–204. Springer, Heidelberg (2013)
8. Wang, C., de Groot, M.: Managing end-user preferences in the smart grid. In: Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking, e-Energy 2010, pp. 105–114. ACM (2010)

May-Happen-in-Parallel Based Deadlock Analysis for Concurrent Objects*

Antonio E. Flores-Montoya[1], Elvira Albert[2], and Samir Genaim[2]

[1] Technische Universität Darmstadt (TUD), Germany
[2] Complutense University of Madrid (UCM), Spain

Abstract. We present a novel deadlock analysis for concurrent objects based on the results inferred by a points-to analysis and a may-happen-in-parallel (MHP) analysis. Similarly to other analysis, we build a *dependency graph* such that the absence of cycles in the graph ensures deadlock freeness. An MHP analysis provides an over-approximation of the pairs of program points that may be running in parallel. The crux of the method is that the analysis integrates the MHP information within the dependency graph in order to discard unfeasible cycles that otherwise would lead to false positives. We argue that our analysis is more precise and/or efficient than previous proposals for deadlock analysis of concurrent objects. As regards accuracy, we are able to handle cases that other analyses have pointed out as challenges. As regards efficiency, the complexity of our deadlock analysis is polynomial.

1 Introduction

The actor-based paradigm [1] on which concurrent objects are based has evolved as a powerful computational model for defining distributed and concurrent systems. In this paradigm, actors are the universal primitives of concurrent computation: in response to a message, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received. The underlying concurrency model of actor languages forms the basis of the programming languages Erlang [3] and Scala [9] that have recently gained in popularity, in part due to their support for scalable concurrency. There are also implementations of actor libraries for Java.

Concurrent objects are actors which communicate via *asynchronous* method calls. Each concurrent object is a monitor and allows at most one *active* task to execute within the object. Scheduling among the tasks of an object is cooperative (or non-preemptive) such that a task has to release the object lock explicitly. Each object has an unbounded set of pending tasks. When the lock of an object is free, any task in the set of pending tasks can grab the lock and

* This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *H*ATS project, and by the Spanish projects TIN2008-05624, TIN2012-38137, PRI-AIBDE-2011-0900 and S2009TIC-1465 *PROMETIDOS-CM*.

start to execute. The synchronization between the caller and the callee methods is performed when the result is strictly necessary. So-called *future variables* are used to decouple method invocation and returned value [6]. The access to values of future variables may require *blocking* the object and waiting for the value to be ready. Thus, blocking and non-blocking asynchronous calls coexist in our framework.

In general, deadlock situations are produced when a concurrent program reaches a state in which one or more tasks are waiting for each other termination and none of them can make any progress. In the concurrent objects paradigm, the combination of non-blocking and blocking mechanisms to access futures may give rise to complex deadlock situations and a rigorous formal analysis is required to ensure deadlock freeness. Similarly to other approaches, our analysis is based on constructing a *dependency graph* which, if acyclic, guarantees that the program is deadlock free. The construction of the graph is done by adding three types of edges between tasks and objects: (1) *task-task* dependency: it indicates that a task is waiting to get a future that another task has to calculate, (2) *task-object* dependency: a task is waiting to get its object's lock, (3) *object-task* dependency: a task is waiting for a future while holding the lock of the object and therefore, making the whole object wait. These dependencies capture all possible deadlock situations that might occur in concurrent objects.

In order to construct the dependency graph, we first perform a points-to analysis [13] which identifies the set of objects and tasks created along any execution. Given this information, the construction of the graph is done by a traversal of the program in which we detect the above types of dependencies. However, without further *temporal* information, our dependency graphs would be extremely imprecise. The crux of our analysis is the use of a precise may-happen-in-parallel (MHP) analysis [2]. Essentially, we label the dependency graph with the program points of the synchronization instructions that introduce the dependencies and, thus, that may potentially induce deadlocks. In a post-process, we discard *unfeasible* cycles in which the synchronization instructions involved in the circular dependency may not happen in parallel. We also describe several extensions to the basic framework to: (1) improve the accuracy of programs that create objects (or tasks) inside loops that are challenging for deadlock analysis, (2) handle concurrent object *groups* and (3) allow the use of future variables as fields. We have implemented our analysis and applied it to formally prove deadlock freeness on an industrial case study developed by Fredhopper®.

2 Language

As in the actor-model, the main idea is that control and data are encapsulated within the notion of concurrent object. This section presents the syntax and semantics of the concurrent objects language, which is basically the same as [11,8,4]. A *program* consists of a set of classes, each of them can define a set of fields, and a set of methods. The notation $\bar{T}$ is used as a shorthand for $T_1, ...T_n$, and similarly for other names. The set of types includes the classes and the set of *future* variable types $fut(T)$. Primitive types and pure expressions pu are omitted for simplicity. The abstract syntax of class declarations CL, method declarations M, types T, variables V, and statements s is:

$$CL ::= \textbf{class}\ C\ \{\bar{T}\ \bar{f}; \bar{M}\} \quad M ::= T\ m(\bar{T}\ \bar{x})\{s; \textbf{return}\ p;\} \quad V ::= x \mid \textbf{this}.f$$
$$s ::= s; s \mid x = e \mid V = x \mid \textbf{await}\ V? \mid \textbf{if}\ p\ \textbf{then}\ s\ \textbf{else}\ s \mid \textbf{while}\ p\ \textbf{do}\ s$$
$$e ::= \textbf{new}\ C(\bar{V}) \mid V!m(\bar{V}) \mid V.\textbf{get} \mid pu \quad T ::= C \mid fut(T) \mid Unit$$

Observe that each object encapsulates a *local heap* which is not accessible from outside this object, i.e., fields are always accessed using the **this** object, and any other object can only access such fields through method calls. We assume that every method ends with a return instruction. We also assume that the program includes a method called main without parameters, which does not belong to any class and has no fields, from which the execution will start. The concurrency model is as follows. Each object has a lock that is shared by all the tasks that belong to the object. Data synchronization is by means of future variables: An **await** $y?$ instruction is used to synchronize with the result of executing task $y{=}x!m(\bar{z})$ such that the **await** $y?$ is executed only when the future variable y is available (i.e., the task is finished). In the meantime, the object's lock can be released and some other *pending* task on that object can take it. In contrast, the expression $y.$**get** blocks the object (no other task of the same object can run) until y is available, i.e., the execution of $m(\bar{z})$ on x is *finished.*

W.l.o.g, we assume that all methods in a program have different names. As notation, we use $body(m)$ for the sequence of instructions defining method m.

2.1 Operational Semantics

A *program state* S is a set $S = \mathtt{Ob} \cup \mathtt{T}$ where Ob is the set of all created objects and, and T is the set of tasks (including active, pending and finished tasks). The associative and commutative union operator on states is denoted by white-space. An *object* is a term $ob(o, a, lk)$ where o is the object identifier, a is a mapping from the object fields to their values, and lk the identifier of the *active task* that holds the object's lock or $\bot$ if the object's lock is free. Only one task can be *active* (running) in each object and has its *lock*. All other tasks are *pending* to be executed, or *finished* if they terminated and released the lock. A *task* is a term $tsk(t, m, o, l, s)$ where t is a unique task identifier, m is the method name executing in the task, o identifies the object to which the task belongs, l is a mapping from local (possibly future) variables to their values, and s is the sequence of instructions to be executed or $s = \epsilon(v)$ if the task has terminated and the return value v is available. Created objects and tasks never disappear from the state.

The execution of a program starts from the initial state $S_0 = \{obj(0, f, 0)$ $tsk(0, \mathsf{main}, 0, l, body(\mathsf{main}))\}$ where we have an initial object with identifier 0 executing task 0. f is an empty mapping (since main had no fields), and l maps local reference and future variables to **null** (standard initialization). The execution proceeds from S_0 by applying *non-deterministically* the semantic rules depicted in Fig. 1 (the execution of sequential instructions is standard and thus omitted). The operational semantics is given in a rewriting-based style where a step is a transition of the form $a\ \underset{\cdots}{\mathrm{b}} \to \underset{\cdots}{\mathrm{b}'}\ c$ in which: dotted underlining indicates that term b is rewritten to b′; we look up the term a but do not modify it and hence it is not included in the subsequent state; and term c is newly added to the state. Transitions are applied according to the rules as follows.

New_Object: an active task t in object o creates an object o' of type B, its fields are initialized with default values (init_atts) and o' is introduced to the state with a free lock. Observe that as the previous object o is not modified, it is not included in the resulting state. Activate: A non finished task can obtain its object's lock if it is free. Async_Call: A method call creates a new task (the initial state is created by *buildLocals*) with a fresh task identifier t_1 which is associated to the corresponding future variable y in l'. Await1: If the future variable we are awaiting for points to a finished task, the await can be completed. The finished task t_1 is only looked up but it does not disappear from the state as its return value may be needed later on. Await2: Otherwise, the task yields the lock so that any other task of the same object can take it. Return: When **return** is executed, the return value is stored in v so that it can be obtained by the future variables that point to that task. Besides, the lock is released and will never be taken again by that task. Consequently, that task is *finished* (marked by adding the instruction $\epsilon(v)$). Get: A $x = y$.**get** instruction waits for the future variable but without yielding the lock. Then, it stores the value associated with the future variable y in x.

3 The Notion of Deadlock

In this section, we formalize the notion of deadlock in concurrent objects in the context of our language. Our notion of deadlock is equivalent to the extended deadlock of [4], which corresponds to the classical definition of deadlock by [10]. As in [4], we distinguish two situations which are instrumental to define the notion of deadlock: a *waiting task* which might be waiting to obtain the lock of the object, or that it is waiting to read a future (using **await** or **get**) that other task has to calculate, and a *blocking task* which is waiting to read a future *and* holds the lock, i.e., using **get**. We refer to the object in which a blocking task is executing as *blocked object*. Possible deadlocks will appear as different combinations of the two synchronization primitives in our language, **await** y? and y.**get**. Detecting deadlocks in a language using such mechanisms is challenging because of potential inconsistencies between synchronization points in separate, yet cooperating, methods, as we show in the following example.

Example 1. Fig. 2 defines several classes (A, B, C, Cl and Sr) with methods that feature typical synchronization patterns. For simplicity, we omit local variables declarations and, at some points, we do not assign the result of y.**get** or method calls to any variable. We illustrate the following types of deadlock: **Selflock** (main1). In this case, there is only object a which introduces a selflock. The call at L3 to blk1 performs a call to empt on the same object a and gets blocked at

$$\text{(New-Object)}\quad \frac{\text{fresh}(o')\ ,\ l' = l[x \to o'],\ a' = \text{init\_atts}(B, \overline{z})}{ob(o,a,t)\ \underline{tsk(t,m,o,l,\{x = \text{new } B(\overline{z}); s\})} \to \underline{tsk(t,m,o,l,s)}\ ob(o',a',\bot)}$$

$$\text{(activate)}\quad \frac{s \neq \epsilon(v)}{\underline{ob(o,a,\bot)}\ tsk(t,m,o,l,s) \to \underline{ob(o,a,t)}}$$

$$\text{(Async-Call)}\quad \frac{l(x) = o_1,\ o_1 \neq \textbf{null},\ \text{fresh}(t_1),\ l' = l[y \to t_1],\ l_1 = buildLocals(\bar{z}, m)}{ob(o,a,t)\ \underline{tsk(t,m,o,l,\{y = x!m_1(\overline{z}); s\})} \to \underline{tsk(t,m,o,l',s)}\ tsk(t_1,m_1,o_1,l_1,body(m_1))}$$

$$\text{(await1)}\quad \frac{l(y) = t_1}{ob(o,a,t)\ \underline{tsk(t,m,o,l,\{\textbf{await } y?; s\})}\ tsk(t_1,m_1,o_1,l_1,\epsilon(v)) \to \underline{tsk(t,m,o,l,s)}}$$

$$\text{(await2)}\quad \frac{l(y) = t_1, s_1 \neq \epsilon(v)}{\underline{ob(o,a,t)}\ tsk(t,m,o,l,\{\textbf{await } y?; s\})\ tsk(t_1,m_1,o_1,l_1,s_1)) \to \underline{ob(o,a,\bot)}}$$

$$\text{(return)}\quad \frac{v = l(x)}{\underline{ob(o,a,t)}\ \ \underline{tsk(t,m,o,l,\{\textbf{return } x; s\})} \to \underline{ob(o,a,\bot)}\ \underline{tsk(t,m,o,l,\epsilon(v))}}$$

$$\text{(get)}\quad \frac{l(y) = t_1,\ l' = l[x \to v]}{ob(o,a,t)\ \underline{tsk(t,m,o,l,\{x{=}y.\textbf{get}; s\})}\ tsk(t_1,m_1,o_1,l_1,\epsilon(v)) \to \underline{tsk(t,m,o,l',s)}}$$

Fig. 1. Summarized semantics of concurrent objects

L27 waiting for its result. The call to empt on a will never be executed. Here, task blk1 is blocking, task empt is a waiting task, and the object a is blocked. **Mutual lock** (main2). Two objects a and b are created. They both execute task blk1 which makes a call to the other object and waits for the result at L27 without releasing the lock. If both tasks execute in parallel, there will be a deadlock (the two tasks are blocking and the objects are blocked). **Mutual indirect** (main3). Object a starts to execute blk2 and object b will execute blk3. In b, blk3 calls wait on c. Then, b gets blocked until wait on c finishes. Now, wait on c calls empt on a and waits for its termination without holding the lock of c (waiting task). If, in the meantime, blk2 was executing in a and called empt on b blocking object a, then empt will never start to execute as b is blocked in blk3. In summary, objects a, b are blocked, tasks blk2 and blk3 are respectively responsible for blocking the objects, task wait on c and task empt on a are waiting tasks. **MHP** (main4). There is no deadlock in the execution of main4 since it is guaranteed that the execution of acc in L66 will start only after the execution of go at L64 has finished. In particular, when the execution of acc blocks the srv object at L57 waiting for termination of rec it is guaranteed that srv is no longer blocked. We will see that the inference of this information requires the enhancement of the analysis with temporal MHP information.

The following state dependencies are equivalent to the notion of extended deadlock defined in [4] and classical deadlock [10], but adapted to our syntax.

```
main1() {
  a=new A();
  a!blk1(a);
}
main2() {
  a=new A();
  b=new A();
  a!blk1(b);
  b!blk1(a);
}
main3() {
  a=new A();
  b=new B();
  c=new C();
   b!blk3(c,a);
   a!blk2(b);
}

main4() {
  Sr s=new Sr();
  s!go();
}

class A {
 Unit blk1(A a) {
   f=a!empt();
   f.get;
 }
 Unit blk2(B b) {
   f=b!empt();
   f.get;
 }
 Unit empt() {}
}
class B {
 Unit blk3(C c,A a) {
   f=c!wait(a);
   f.get;
 }
 Unit empt() {}
}

class C {
 Unit wait(A a) {
   f=a!empt();
   await f?;
 }
}

class Cl {
 Sr srv;
 Unit go(Sr s) {
   srv=s;
 }
Unit acc() {
  f=srv!rec("...");
  f.get;
 }
}
class Sr {
  Unit rec(Str m){}
  Unit go() {
   c=new Cl();
  f=c!go(this);
  f.get;
  c!acc();
 }
}
```

Fig. 2. Simple examples featuring different types of deadlock

Definition 1 (state dependencies). *Given a program state $S = \mathtt{Ob} \cup \mathtt{T}$, we define its dependency graph G_S whose nodes are the existing object and task identifiers and whose edges are defined as follows:*

1. ***Object-Task:*** *$o \to t_2$ iff there is an object $obj(o, a, t) \in \mathtt{Ob}$ and tasks $tsk(t, m, o, l, \{x = y.\mathbf{get}; s\})$, $tsk(t_2, m, o_2, l_2, s_2) \in \mathtt{T}$ where $l(y) = t_2$ and $s_2 \neq \epsilon(v)$.*
2. ***Task-Task:*** *$t_1 \to t_2$ iff there are two tasks $tsk(t_1, m_1, o_1, l_1, \{sync; s\})$, $tsk(t_2, m_2, o_2, l_2, s_2) \in \mathtt{T}$ where $sync \in \{x = y.\mathbf{get}, \mathbf{await}\ y?\}$, $l_1(y) = t_2$ and $s_2 \neq \epsilon(v)$.*
3. ***Task-Object:*** *$t \to o$ iff there is a task $tsk(t, m, o, l, s) \in \mathtt{T}$ and an object $obj(o, a, lk) \in \mathtt{Ob}$ with $lk \neq t$ and $s \neq \epsilon(v)$.*

The first type of dependency corresponds to the notion of blocking task and blocked object and the other two to waiting tasks. Dependencies are created as long as the task we are waiting for is not finished. Observe that a **get** instruction will generate two dependencies, whereas an **await** will generate only a dependency. Besides, every task without the object's lock (which is not finished) has a dependency to its object.

Example 2. Let us consider the final (deadlock) state for main3 described in Ex. 1. Here, we denote by o:m a task executing method m on object o. We have the following seven dependencies in this state which form a cycle:

d1	a $\to$ b:empt	d3	a:empt $\to$ a	d5	b $\to$ c:wait	d7	c:wait $\to$ a:empt
d2	a:blk2 $\to$ b:empt	d4	b:blk3 $\to$ c:wait	d6	b:empt $\to$ b		

Observe that in object a we have a blocking task a:blk2 executing a **get** which induces dependencies d1 and d2 above, and a waiting task a:empt that induces d3. In b, we have the blocking task b:blk3 that adds d4 and d5 and a waiting task b:empt that adds d6. In c, we have a waiting task c:wait that induces d7.

Definition 2 (deadlock). *A program state S is deadlock iff its dependency graph G_S contains a cycle.*

We are assuming that object fields cannot contain future variables. The removal of this restriction will be discussed in Sec. 5.2. As a consequence, there cannot be cycles involving only task-task dependencies. Intuitively, a cycle that involves only task-task dependencies represents a task that is (possibly indirectly) waiting for itself. Without future fields, a task t can only wait for: other tasks that were created strictly before but did not call t (using futures as parameters); or tasks that were called (possibly indirectly) by t itself. t has no access to its future nor to any of these tasks. Consequently, at least one object must be involved in a cyclic dependency. Additionally, **await** y? instructions only create task-task dependencies. In order to have cycles, we need at least one object-task dependency. Therefore, at least one y.**get** instruction must be involved.

4 Deadlock Analysis

In this section we describe our deadlock analysis which over-approximates the notion of deadlock in Def. 1. If the analysis reports that a program is deadlock-free, then there is no execution that reaches a deadlock state. When the analysis reports a *potential* deadlock, it also provides hints on the program points involved in this deadlock. Our analysis performs two steps: (1) We generate an *abstract* dependency graph $\mathcal{G}$ that over-approximates the dependency graphs G_S of any reachable state S. This graph is obtained by abstracting objects and tasks using points-to analysis. (2) We declare every cycle in $\mathcal{G}$ as a potential deadlock scenario and, in a post-process, we eliminate unfeasible scenarios by discarding those cycles whose involved program points cannot execute in parallel. For the latter, we rely on an MHP analysis. In Sec. 4.1, we describe how points-to analysis is used to abstract objects and tasks. In Sec. 4.2, we present our notion of abstract dependency graph.

4.1 Abstract Tasks and Abstract Objects

Abstracting objects is an extensively studied problem in program analysis. It is at the heart of almost any static analysis for object oriented programs, and usually is referred to as points-to analysis [13]. In principle, any points-to analysis can be used to obtain the information we require. The choice, however, affects the performance and precision of our deadlock analysis. In what follows, we explain how we use *object-sensitive* [13] points-to analysis in order to abstract not only objects, but also tasks. An analysis is object-sensitive if methods are analyzed separately for the different (sets of) objects on which they are invoked. As objects are the concurrency units, object-sensitive points-to analysis naturally suits our setting since tasks are identified with the objects on which they execute.

Following [13,15], objects are abstracted to syntactic constructions of the form $ob_{ij\ldots pq}$, where all elements in $ij\ldots pq$ are allocation sites (the program points in which objects are created). The abstract object $ob_{ij\ldots pq}$ represents all run-time objects that were created at q when the enclosing instance method was invoked on an object represented by $ob_{ij\ldots p}$, which in turn was created at allocation site p. As notation, we let A be the set of all allocation sites, $\mathcal{P}$ be the set of program points, $\mathcal{V}$ be the set of variables and $pp(s)$ be the program point where statement s is. Given a constant $k \geq 1$, the analysis computes (i) a finite set of abstract object names $\mathcal{O} \subseteq \{ob_\ell \mid \ell \in A \cup A^2 \cup \cdots \cup A^k\}$; and (ii) a partial function $\mathcal{A} : \mathcal{O} \times \mathcal{P} \times \mathcal{V} \mapsto \wp(\mathcal{O} \cup \{\mathbf{null}\})$, where $\mathcal{A}(ob, p, x)$ is the set of abstract objects to which the *reference variable* x might point to, when executing program point p on the abstract object ob. Constant k defines the maximum length of allocation sequences, and it allows controlling the precision of the analysis and ensuring termination. Allocation sequences may have unbounded length and thus it is sometimes necessary to approximate such sequences. This is done by just keeping the k rightmost positions in sequences whose length is greater than k.

We also use the points-to information for task abstraction. Intuitively, we let $\mathcal{T} = \{ob.m \mid ob \in \mathcal{O},\ m \text{ is a method name}\}$ be the set of abstract task identifiers, where $ob.m \in \mathcal{T}$ represents a task that executes the code of method m on the abstract object ob. The points-to analysis is modified to track the values of future variables (which are task identifiers). We distinguish two kinds of task identifiers: normal tasks $tk \in \mathcal{T}$, whose result might be available or not; and ready tasks $tk_r \in \mathcal{T}_r$, whose result is guaranteed to be available (and therefore will not cause any further waiting). Briefly, the significant changes are: (i) the analysis of $y = x!m(\bar{z})$, in which y is assigned the set of abstract task identifiers tk that are induced by the abstract objects to which variable x points-to; and (ii) the analysis of y.**get** and **await** $y?$, in which each $tk \in \mathcal{A}(ob, p, y)$ is substituted by tk_r. In order to use this information, we abuse notation and assume that function $\mathcal{A}$ is extended to map future variables to elements of $\wp(\mathcal{T} \cup \mathcal{T}_r)$. We use function α to denote the mapping from concrete object and task identifiers to corresponding abstract ones.

Example 3. Let us consider the analysis of method main2. The objects created at L6 and L7 are abstracted to ob_6 and ob_7 respectively. Thus, the tasks spawned at L8 and L9 are abstracted to ob_6.blk1 and ob_7.blk1, respectively. Within the two executions of blk1, new tasks executing empt are spawned. They are executed on the object that is passed as parameter. Hence, we keep two separate abstractions, ob_6.empt for the task executing on ob_6, and ob_7.empt for the one executing on ob_7. Next, the future variable f is assigned the abstract value of the tasks whose termination is waiting for. Thus, the value of f is abstracted to ob_7.empt for the task executing on ob_6 and to ob_6.empt for the one executing on ob_7. Note that the use of object-sensitive information is fundamental for precision. Using object-insensitive analysis, all calls to the methods blk1 and empt had been abstracted by the same abstract task identifier (instead of keeping the two identifiers separate), and thus had led to an utter lose of precision.

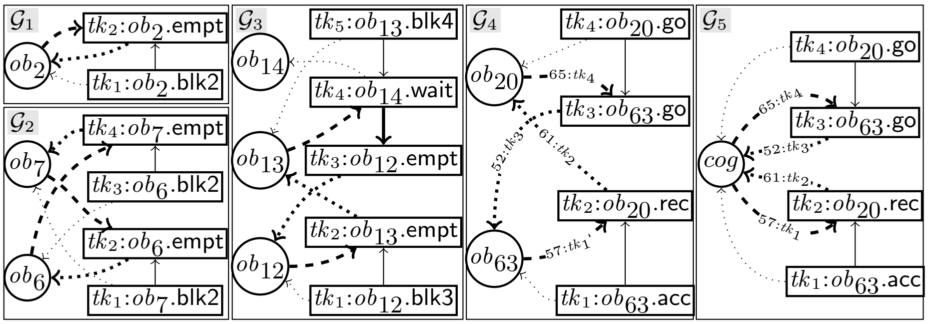


Fig. 3. ($\mathcal{G}_1 - \mathcal{G}_4$) abstract dependency graphs for main1-main4 of Fig. 2; ($\mathcal{G}_5$) Abstract dependency graphs for main4 with COGs;

4.2 Abstract Dependency Graph

Given the object and task abstractions provided in Sec. 4.1, we can construct an *abstract dependency graph* for the program as follows.

Definition 3 (abstract dependency graph). *The abstract dependency graph is a directed graph $\mathcal{G}$, whose nodes are $\mathcal{O} \cup \mathcal{T}$, and whose edges are:*

1. ***Object-Task**: $ob \xrightarrow{p:tk} tk'$ iff there is an instruction $x = y.$**get** at program point $p \in \mathcal{P}$, $tk, tk' \in \mathcal{T}$ and $ob \in \mathcal{O}$ such that $tk = ob.m$ and $tk' \in \mathcal{A}(ob, p, y)$.*
2. ***Task-Task**: $tk \xrightarrow{p:tk} tk'$ iff there is an instruction $x = y.$**get** or **await** $y?$ at program point $p \in \mathcal{P}$, $tk, tk' \in \mathcal{T}$ such that $tk = ob.m$ and $tk' \in \mathcal{A}(ob, p, y)$.*
3. ***Task-Object**: $tk \xrightarrow{p:tk} ob$ iff $tk \in \mathcal{T}$ and $ob \in \mathcal{O}$ such that $tk = ob.m$, where p is the entry program point of m, or of an instruction **await** $y?$ in m.*

Observe that the construction follows exactly Def. 1, but we use abstract information instead of the concrete one. The nodes are the abstract objects and tasks, and the edges represent the following information: (1) the abstract object ob is locked by the abstract task tk until task tk' finishes; (2) the abstract task tk is waiting for the abstract task tk' to finish; and (3) the abstract task tk might be waiting on the abstract object ob. The labels on the edges $p{:}tk$ keep information on the source of this edge, p is a program point in the task tk. These labels will be used below. Roughly, the abstract dependency graph can be seen as an abstraction of the graph that results from the union of all G_S, where some nodes are collapsed. Note that ready tasks tk_r are ignored as they cannot involve any waiting.

Example 4. Fig. 3 shows the abstract dependency graphs obtained from the analysis of the four main methods in Fig. 2 ($\mathcal{G}_5$ will be explained later). Cycles are marked with bold edges. The deadlocks informally described in Ex. 1 for the first three main methods can be seen in the graphs. We have omitted the labels in these three graphs as they are not relevant. For instance, in $\mathcal{G}_3$, the cycle includes the two blocked objects a (here ob_{12}) and b (here ob_{13}) and the three waiting tasks as described in Ex. 1. An important point to note is that $\mathcal{G}_4$ contains a cycle. However, as justified informally in Ex. 1, the program is deadlock free. The problem is that the graph does not contain *temporal* information about whether the instructions involved in the cycle may indeed happen in parallel.

The MHP analysis of [2] is adapted to infer a set of symmetric pairs $\mathcal{M} \subseteq ((\mathcal{P} \times \mathcal{T}) \times (\mathcal{P} \times \mathcal{T}))$ with the following guarantee: for any reachable state S, if $tsk(t_1, m_1, o_1, l_1, s_1)$ and $tsk(t_2, m_2, o_2, l_2, s_2)$ are two tasks of S available in the state such that $t_1 \neq t_2$, then $(p_1{:}tk_1, p_2{:}tk_2) \in \mathcal{M}$ where $p_i = pp(s_i)$ and $tk_i = \alpha(t_i)$. Intuitively, if program points p_1 and p_2 might execute in parallel within tasks t_1 and t_2, then $\mathcal{M}$ includes this information at the level of the corresponding abstract tasks.

Example 5. As an example, the application of the MHP [2] to main2 gives, among others, the MHP pair $(27{:}tk_3, 27{:}tk_4)$ where tk_3 and tk_4 are shortcuts given in Fig. 3 for ob_6:blk1 and ob_7:blk1, respectively. This pair indicates that objects ob_6 and ob_6 can be executing the **get** instruction at program point 27 in parallel. Thus, a deadlock would occur. The MHP of main4 gives the following set of MHP pairs $\{(61{:}tk_2, 57{:}tk_1), (65{:}tk_4,\ 52{:}tk_3)\}$. The important point to notice is that instructions $65{:}tk_4$ and $57{:}tk_1$ cannot happen in parallel. This formally justifies the intuition for deadlock freeness given in Ex. 1.

Definition 4 (feasible cycle). *A cycle* $e_1 \xrightarrow{p_1:tk_1} e_2 \xrightarrow{p_2:tk_2} \ldots \xrightarrow{p_n:tk_n} e_1 \in \mathcal{G}$ *is feasible iff* $(p_i{:}tk_i, p_j{:}tk_j) \in \mathcal{M}$ *for all* $1 \leq i < j \leq n$*, and at least one* e_i *is an object identifier.*

As we have mentioned before, a deadlock must involve at least one object which is blocked. Also, all points that are involved in the considered cycle must happen in parallel (this is the condition over-approximated by MHP). In the implementation, instead of first building the dependency graph and then checking the feasibility of cycles, for each cycle, there is an interleaved construction such that new dependencies are only added if they satisfy the MHP condition with respect to the previous ones.

Example 6. The cycle in $\mathcal{G}_5$ is not feasible because $(65 : tk_4, 57 : tk_1)$ does not belong to the MHP pairs given in Ex. 5.

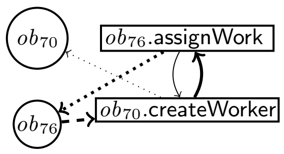
Our soundness theorem ensures that if there is a deadlock in the execution of the concrete program, then the abstract graph contains a feasible cycle.

Theorem 1 (soundness). *Let* S *be a reachable state. If there is a cycle* $e_1 \to e_2 \to \cdots \to e_1$ *in* G_S*, then* $\alpha(e_1) \xrightarrow{p_1:tk_1} \alpha(e_2) \xrightarrow{p_2:tk_2} \cdots \xrightarrow{p_n:tk_n} \alpha(e_1)$ *is a feasible cycle of* $\mathcal{G}$*.*

The following corollary follows trivially from the above theorem.

Corollary 1 (deadlock-freeness). *If there are no feasible cycles in* $\mathcal{G}$*, then the program is deadlock-free.*

```
69 main(Int n) {
70   c=new Factory();
71   f=c!createWorker(n);
72   await f?;
73 }
```



```
74 class Factory() {
75   Unit createWorker(Int n){
76     w = new Worker(this);
77     f=w!assignWork(n);
78     await f?;
79   }
80 }
```

```
81 class Worker(Factory fc) {
82   Unit assignWork(Int n) {
83     if(n>0) {
84       f=fc!createWorker(n−1);
85       f.get;
86     }
87   }
88 }
```

Fig. 4. Challenging example of [8] for handling objects created in loops

5 Extensions of the Basic Framework

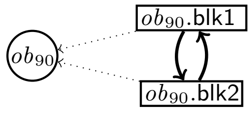
In this section, we present several extensions of our analysis in order to improve the precision on some challenging examples, as well as extensions to handle advanced features such storing future variables in fields, and objects groups.

5.1 Creating Objects and Tasks Inside Loops

Programs that create objects (or tasks) inside loops are challenging for deadlock analysis due to the fact that the created objects, whose amount usually depends on an unknown input value, are abstracted to a finite set of abstract objects (or tasks). This makes it difficult to distinguish their activities and leads to spurious scenarios. The example in Fig. 4 is a program reported in [8] as challenging (they failed prove it deadlock free). When calling main(n), the program creates n objects of type Worker using a single Factory object. The first call createWorker(n) creates one object of type Worker, calls its assignWork method, and waits until this call finishes. Method assignWork in turn calls createWorker(n−1) in order to create the remaining n−1 objects, and waits (blocking the current Worker object) until this call finishes. This program is deadlock free, because every Worker object waits (transitively) for a task that is running on a *different* Worker object. However, our basic framework as described in Sec. 4 generates the abstract dependency graph shown in the figure, which contains a cycle. Node ob_{76} represents all objects of type Worker, and ob_{76}.assignWork represents all tasks of method assignWork.

The cycle represents a scenario in which an object of type Worker (the source object) is blocked waiting (transitively) for another task running on an object of type Worker (the target object) to finish. Since both the source and target objects are represented by the same abstract value ob_{76}, we have to assume the case in which they are equal, and thus create a deadlock. This, however, is a spurious scenario that cannot actually happen. Our aim is to prove that this cycle is unfeasible, in particular that the source and target objects cannot be the same even if they are assigned the same abstract value. Consider the dependency ob_{70}.createWorker $\rightarrow$ ob_{76}.assignWork, and observe that whenever createWorker calls

```
89 main() {
90   A a = new A();
91   a!run();
92 }
```



```
93 class A {
94   Fut<Unit> f;
95   Bool ready=False;
96   Unit run(){
97     ff=this!blk1();
98     f=this!blk2(ff);
99     ready=True;
100  }
101  Unit blk1(){
102    await ready==True;
103    await f?;
104  }
105  Unit blk2(Fut<Unit> ff){
106    await ff?;
107  }
108 }
```

Fig. 5. Deadlock Analysis for of Future Variables as Fields

assignWork it uses a *fresh* object, i.e., an object that has been created in its local scope. Thus, it cannot be the case that assignWork belongs to the same object that is blocked waiting for createWorker to finish. This simple information is enough to discard this cycle. Our extension is based on identifying edges with this *freshness* property, and then using them to discard cycles as the one above, and several other cases. The appealing feature of the *freshness* property is that it can inferred with almost no further overhead. Briefly, it is done by modifying the points-to analysis to annotate abstract objects with a *freshness flag*, which indicates that it represents a single object that has been recently created in the current scope. Once objects *escape* from their local scope the corresponding flags are discarded.

5.2 Future Variables as Fields

So far we have assumed that future variables are passed (by-value) between methods only as parameters or return values. This restriction guarantees that any deadlock must involve at least one object that is locked by one of its tasks (using instruction y.**get**). In this section, we allow future variables to be defined as fields. This is challenging because a task gains access to its identifier. Consider the program in Fig. 5. It is easy to verify that, starting from method main, we can reach a state in which tasks ob_{90}.blk1 and ob_{90}.blk2 are waiting for each other: ob_{90}.blk2 receives the task identifier of ob_{90}.blk1 as parameter, while ob_{90}.blk1 reads that of ob_{90}.blk2 from a field. By applying our analysis, we get the graph shown in Fig. 5, which does not have any cycle that goes through an object and hence, according to Def. 4, we incorrectly conclude that it is deadlock free. The reason is that we cannot ignore cycles that involve only task-task nodes. Thus, Def. 4 should be modified to drop the requirement that the cycle includes at least one object node.

This change only affects the latter part of the analysis where we explore the abstract dependency graph and the rest of the analysis is still valid. However, we can expect an increment on the number of false positives. Many recursive methods will induce new cycles such as the example in Sec. 5.1: we have to consider the cycle that only involves assignWork and createWorker. Fortunately, we can apply the notion of freshness described in Sec. 5.1 to tasks rather than to objects and discard most of these new false positives. Briefly, the *task-freshness* property can be used to prove that a task always waits for other (fresh) tasks that have been created in its local scope, and thus it cannot be the case that a task is transitively waiting for itself to finish.

5.3 Object Groups

Object *groups* extend the concurrent objects model to allow grouping objects into concurrency units such that at most one task can be executing among all objects in each group. At the programming language level, this feature is supported by the instruction **new cog** $C(\bar{x})$ which creates a new object of type C and, in addition, assigns it to a *new* group that includes this single object. The instruction **new** $C(\bar{x})$, in turn, creates a new object of type C and assigns it to the group of its parent object (i.e, the object that executed the instruction). The definition of deadlock in this setting is very similar to Defs. 1 and 2. The difference is that object nodes are replaced by group nodes, and the ***Object-Task*** and ***Task-Object*** edges connect a task t with the node that corresponds to the group of o. Then, a program deadlocks iff it reaches a state whose dependency graph includes a cycle with at least one group node. Similarly, at the abstract level, the deadlock property can be approximated as in Sec. 4, by replacing abstract object nodes by abstract group nodes. This, however, requires inferring a set of abstract groups and relating them to the set of abstract objects. In practice, we infer this information by modifying the points-to analysis to track the (abstract) group of each abstract object in a straightforward way. This modification has negligible overhead.

Example 7. Assuming a concurrent object groups setting, method main5 is not deadlock free any more. Our analysis infers that objects created at L20 and L63 belong to the same group. Then, it constructs the abstract dependency graph $\mathcal{G}_6$ depicted in Fig. 3. Note that $\mathcal{G}_6$ merges the two object nodes of $\mathcal{G}_5$ into a single group node. Since $\mathcal{G}_6$ contains a cycle $cog \rightarrow ob_{63} \rightarrow cog$, and moreover $65{:}tk_4$ and $52{:}tk_3$ may execute in parallel, it reports a deadlock. By replacing **new** by **new cog** the code would be deadlock free and detected as such by the analysis.

6 Experiments

We report on DECO[1], a DEadlock analyzer for Concurrent Objects, which implements the analysis described in the paper and the extensions for the complete ABS language [11]. Given a program with a main procedure, the output of the analysis is a description of the potential cycles (if any) so that the user can easily find the causes of the deadlocks and discard false positives. This section aims at experimentally evaluating the accuracy and performance of the DECO tool, and comparing our results with those obtained by the SDA tool [7]. However, such

[1] DECO can be tried online at `http://costa.ls.fi.upm.es/costabs/deco`

		SDA Tool		DECO Tool				
Medium codes	Lines	Result	T(sec)	Result	PtT(ms)	MhpT(ms)	DT(ms)	T(sec)
Bookshop	418	$\times$	-	✓	< 20	< 20	< 20	1.36
PeerToPeer	185	$\times$	-	✓	99	359	< 20	1.63
LeaderElection	63	$\times$	-	✓	< 20	< 20	< 20	1.48
PingPong	66	$\times$	-	✓	< 20	< 20	< 20	1.30
MultiPingPong	89	$\times$	-	18	< 20	< 20	< 20	1.43
BoundedBuffer	103	✓	1.65	✓	< 20	< 20	< 20	1.26
Case Studies	Lines	Result	T(sec)	Result	PtT(ms)	MhpT(ms)	DT(ms)	T(sec)
TradingSystem	1466	$\times$	-	✓	79	491	< 20	3.15
FredHopper	2111	$\times$	-	✓	372	2456	351	6.62
Adpt Fredhopper	2081	✓	65.30	✓	136	1347	73	4.38

Fig. 6. Medium examples and case studies results

comparison is not always feasible because SDA does not handle complete ABS. Basically, they have to adapt the programs to remove recursive object structures. Otherwise, SDA fails to obtain any result. They also annotate **await** instructions with boolean conditions to increase the precision. Await boolean conditions are instructions of the form **await** e where e is a boolean expression. This instruction releases the object's lock when the condition is not satisfied. This kind of instructions can be used for internal synchronization within an object. Our analysis supports recursive object structures naturally and we have developed an improvement over the MHP analysis of [2] to take simple **await** instructions with boolean conditions (without function calls) into account. More complex boolean awaits are ignored without harming the soundness of the analysis.

Both tools have been tested on: 39 small examples (19 are taken from [7] and 20 are developed by us); on 6 medium-size programs written for benchmarking purposes by ABS programmers; and two case studies developed by Fredhopper®. The source code of all examples can be found in the above website. All examples were run with a constant $k = 2$ for the poinsto analysis. About the small examples, it is worth mentioning that our tool does not report any false positive, while SDA gives 8 false positives (5 of them are on our examples and 3 are on theirs) and fails to analyze 2 examples. The failures are due to the recursive object structures limitation above. Out of the 8 false positives, one is due to their treatment of loops over data structures and the remaining ones are correctly discarded in our tool thanks to the extension described in Sec. 5.1.

Fig. 6 reports the results on the medium-size benchmarks and on the case studies. The first column contains the name of the benchmarks and the second one its size (number of lines). The leftmost set of columns are the results computed by the SDA tool and the rightmost set by our implementation. In both sets, the first column shows the result of the analysis: $\times$ means that the tool fails to analyze it, ✓means that it proves deadlock freeness, and a positive number meaning the number of cycles found. The next column shows the analysis time in seconds (average of 10 runs). For DECO, we show first the analysis time of the different phases of the analysis: (PtT) is points-to analysis time, (MhpT) is MHP time, (DT) is the time spent on creating and exploring the abstract dependency graph. The rightmost column is the overall time. It can be seen that we have proved deadlock freeness in all medium-size programs except for *MultiPingPong* which reports 18 potential deadlocks. By executing the program step by step we have checked that these potential deadlocks are indeed real deadlocks. SDA was able to analyze only the *BoundedBuffer* example. Analyzing the remaining examples would require rewriting them to avoid recursive object structures.

As regards the case studies, the first two ones are the original versions while the last one is a modification of the second one to avoid the limitations of SDA described above. Our tool can successfully analyze both the original and the modified versions reporting deadlock freeness. As regards performance, all analyses have been performed in an Ubuntu 12.04 64-bit with Intel core i7-3667U 2.00GHz x 4 and 8GiB of Memory. The total times for the two tools have been externally measured, while partial times of our analysis have been measured internally. These times do not take compilation and program initialization into account and thus they do not add to the total time. They also have a limited precision and thus negligible times are often reported as 0. We present them as < 20 in the table. We can see that, for *BoundedBuffer*, both tools have a similar performance (1.65 secs and 1.26 secs). However, when it comes to analyze bigger programs our approach is much more efficient (4.38 secs over 65.30 secs) which seems to indicate that our techniques are more scalable.

7 Related Work and Conclusions

We have presented a novel deadlock analysis for languages with actor-based concurrency based on a points-to analysis and an MHP analysis. We argue that our technique outperforms previous proposals [8,5,4,7] in precision or efficiency and it considers a more expressive language. Experiments suggest we achieve better precision than [8,7] because our pointsto analysis keeps a more fine-grained representation of objects and their dependencies than their contract-based approach. That, together with Sec. 5.1, gives precise results in examples pointed out as challenging in [8,7]. Besides, our analysis does not suffer from any of the restrictions mentioned in Sec. 6. The language used in [5] is very restrictive (e.g., it does not have an object creation instruction, nor synchronizations using await, among other limitations). More recent work [4] improves the previous one, since the use of Petri nets specifies the temporal behavior of the methods. However, the language is again more restrictive than ours since it does not allow dealing with objects stored in fields and does not treat while loops and object creation explicitly. Even more importantly, our analysis is polynomial, while [4] requires solving a reachability problem in Petri nets (which is EXPSPACE-hard). Note that the complexity of the points-to analysis can vary depending on the precision, it is almost linear in [16] and [13] has cubic worst-case complexity.

It is not the first time that an MHP analysis has been used to detect deadlocks. Its use dates back to 1991 [12] where it was applied to detect deadlocks on Ada programs. It has been also applied in [14] to thread-based programs. There are some fundamental differences between [14] and our work due also in part to the differences between the underlying concurrency models. Their algorithm detects locks due to lock-based synchronization whereas wait-notify in Java is not covered. In contrast, we treat wait-notify synchronization and in particular our treatment of future variables (which represent the notify) is very powerful. In their case, the algorithm detects deadlocks between two threads, while our technique does not have this restriction and can detect deadlocks that involve any number of objects. On the other hand, they propose some (unsound) treatment to non-guarded and non-reentrant conditions that target Java programs but cannot happen in our framework.

References

1. Agha, G.A.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
2. Albert, E., Flores-Montoya, A.E., Genaim, S.: Analysis of May-Happen-in-Parallel in Concurrent Objects. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 35–51. Springer, Heidelberg (2012)
3. Armstrong, J., Virding, R., Wistrom, C., Williams, M.: Concurrent Programming in Erlang. Prentice Hall (1996)
4. de Boer, F.S., Bravetti, M., Grabe, I., Lee, M., Steffen, M., Zavattaro, G.: A Petri Net based Analysis of Deadlocks for Active Objects and Futures. In: Păsăreanu, C.S., Salaün, G. (eds.) FACS 2012. LNCS, vol. 7684, pp. 110–127. Springer, Heidelberg (2013)
5. de Boer, F.S., Grabe, I., Steffen, M.: Termination detection for active objects. J. Log. Algebr. Program. 81(4), 541–557 (2012)
6. 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)
7. Giachino, E., Grazia, C.A., Laneve, C., Lienhardt, M., Wong, P.: Deadlock Analysis of Concurrent Objects – Theory and Practice (2013)
8. Giachino, E., Laneve, C.: Analysis of Deadlocks in Object Groups. In: Bruni, R., Dingel, J. (eds.) FORTE 2011 and FMOODS 2011. LNCS, vol. 6722, pp. 168–182. Springer, Heidelberg (2011)
9. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theor. Comput. Sci. 410(2-3), 202–220 (2009)
10. Holt, R.C.: Some Deadlock Properties of Computer Systems. ACM Comput. Surv. 4(3), 179–196 (1972)
11. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
12. Masticola, S.P., Ryder, B.G.: A Model of Ada Programs for Static Deadlock Detection in Polynomial Time. In: Parallel and Distributed Debugging, pp. 97–107. ACM (1991)
13. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for java. ACM Trans. Softw. Eng. Methodol. 14, 1–41 (2005)
14. Naik, M., Park, C., Sen, K., Gay, D.: Effective static deadlock detection. In: Proc. of ICSE, pp. 386–396. IEEE (2009)
15. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: understanding object-sensitivity. In: POPL, pp. 17–30. ACM (2011)
16. Steensgaard, B.: Points-to analysis in almost linear time. In: Symposium on Principles of Programming Languages, pp. 32–41 (1996)

Lintent: Towards Security Type-Checking of Android Applications

Michele Bugliesi, Stefano Calzavara, and Alvise Spanò

Università Ca' Foscari Venezia

Abstract. The widespread adoption of Android devices has attracted the attention of a growing computer security audience. Fundamental weaknesses and subtle design flaws of the Android architecture have been identified, studied and fixed, mostly through techniques from data-flow analysis, runtime protection mechanisms, or changes to the operating system. This paper complements this research by developing a framework for the analysis of Android applications based on typing techniques. We introduce a formal calculus for reasoning on the Android inter-component communication API and a type-and-effect system to statically prevent privilege escalation attacks on well-typed components. Drawing on our abstract framework, we develop a prototype implementation of `Lintent`, a security type-checker for Android applications integrated with the Android Development Tools suite. We finally discuss preliminary experiences with our tool, which highlight real attacks on existing applications.

1 Introduction

Mobile phones have quickly evolved from simple devices intended for phone calls and text messaging, to powerful handheld PDAs, hosting sophisticated applications that manage personal data and interact on-line to share information and access (security-sensitive) services. This evolution has attracted the interest of a growing community of researchers on mobile phone security, and on Android security in particular.

Fundamental weaknesses and subtle design flaws of the Android architecture have been identified, studied and fixed. Originated with the seminal work in [9], a series of papers have developed techniques to ensure various system-level information-flow properties, by means of data-flow analysis [13], runtime detection mechanisms [7] and changes to the operating system [12]. Other papers have applied similar techniques to the study of the intent-based communication model of Android and its interaction with the underlying permission system [5,2]. Somewhat surprisingly, typing techniques have instead received very limited attention, with few notable exceptions to date ([3], and more recently [1]). As a result, the potential extent and scope of type-based analysis has been so far left largely unexplored. In the present paper we make a step towards filling this gap.

Contributions. Our analysis of the Android platform is targeted at the static detection of privilege escalation attacks, a vulnerability which exposes the framework to the risk of unauthorized permission usage by malicious applications.

To carry out our study, we introduce π-`Perms`, a simple formal calculus for reasoning about inter-component interaction in Android (Section 3). Albeit small and abstract, π-`Perms` captures the most relevant aspects of the Android message passing architecture and its relationships with the underlying permission system. Our formalization pays off, as it allows us to unveil subtle attack surfaces to the current Android implementation that had not been evaluated before.

We tackle the problem of programmatically preventing privilege escalation attacks inside π-`Perms`, by spelling out a formal definition of safety (Section 4) and proposing a sound security type system which statically enforces such notion, despite the best efforts of an opponent (Section 5). Providing the desired protection turns out to be challenging, since the inadvertent disclosure of sensitive data may enable some typically overlooked privilege escalation scenarios.

Based on our formal framework, we then develop a prototype implementation of `Lintent`, a type-based analyzer integrated with the Android Development Tools suite (Section 6). `Lintent` integrates our typing technique for privilege escalation detection within a full-fledged static analysis framework aimed at supporting a robust and more reliable development process. `Lintent` is the first type-based analyzer for Android applications and its implementation highlights a number of engineering challenges which should likely be tackled by any other type-based verification tool for Android. We discuss preliminary experiences with our tool, which highlight real attacks on existing applications (Section 7).

Enhancing the Android development process is increasingly being recognized as an urgent need [4,10,8,16,6]: `Lintent` represents a first step in that direction[1].

2 Android Overview

Intents. Once installed on a device, Android applications run isolated from each other in their own security sandbox. Data and functionality sharing among different applications is implemented through a message-passing paradigm built on top of *intents*, i.e., asynchronous messages providing an abstract description of an operation to be performed. Intents may be either *explicit* or *implicit*: the former specify their intended receiver by name and are always securely delivered to it; the latter, instead, do not mention any specific receiver and just require delivery to any application that supports a given operation (an *action*).

Components. Intents are delivered to application *components*, the essential building blocks of Android applications. There are four different types of components. An *activity* represents a screen with a user interface: activities are started with an intent and possibly return a result upon termination. A *service* runs in the background to perform long-running computations: services can either be started with an intent, or expose a remote method invocation interface to a client by returning it a *binder* object. A *broadcast receiver* waits for intents sent to multiple applications. A *content provider* manages a shared set of persistent application data. Content providers are not accessed through intents, but through a CRUD (Create-Read-Update-Delete) interface reminiscent of SQL.

[1] Technical report and `Lintent` at `https://github.com/alvisespano/lintent`

Protection Mechanisms. The Android security model implements isolation and privilege separation on top of a simple permission system. Android permissions are identified by strings and can be defined by either the operating system or the applications. Permissions are assigned at installation time and are shared by all the components of the same application; if any of the requested permissions is not granted by the user, the application is not installed. The Android communication API offers various protection mechanisms to the different component types. In particular, all components may declare permissions which must be owned by other components requesting access to them; on the other hand, only broadcast requests may specify a permission which a receiver must hold to get the message. A limited form of permission delegation is implemented in Android by special objects known as *pending intents*: we will return to this point later on.

3 π-Perms: A Calculus for Android Applications

We describe π-Perms, a simple formal calculus which captures the essence of inter-component communication in Android. We detail the connections between π-Perms and the Android platform in Section 3.2.

3.1 Syntax and Semantics

We presuppose disjoint collections of names m, n and variables x, y, z, and use the meta-variables u, v to range over *values*, i.e., both names and variables. We denote permissions with typewriter capital letters, as in PERMS, and assume they form a complete lattice with partial order $\sqsubseteq$, top and bottom elements $\top$ and $\bot$ respectively, and join and meet operators $\sqcup$ and $\sqcap$ respectively.

An *expression* represents a sequential program, which runs with a given set of assigned permissions and may return a value. As part of its computation, an expression may perform function calls from a pool of *function definitions*. The syntax of expressions is defined in Table 1.

Table 1. Syntax of π-Perms expressions

$E ::=$	*expressions*	$D ::=$	*definitions*
$D \setminus E$	evaluation	$u(x \triangleleft \texttt{CALL}).E$	function def.
$\overline{u}\langle v \triangleright \texttt{RECV}\rangle$	invocation	$D \wedge D$	conjunction
$\mathsf{let}\ x = E\ \mathsf{in}\ E'$	let expr.		
$(\nu n)\, E$	restriction		
$[\texttt{PERMS}]\, E$	perm. assign.		
v	value		

The expression $D \setminus E$ runs E in the pool of function definitions D. An invocation $\overline{u}\langle v \triangleright \texttt{RECV}\rangle$ tries to call function u, supplying v as an argument; the invocation succeeds only if the callee has at least permissions RECV. A let expression $\mathsf{let}\ x = E\ \mathsf{in}\ E'$ evaluates E to a name n and then behaves as E' with x substituted by n. A restriction $(\nu n)\,E$ creates a fresh name n and then behaves as E. The expression $[\texttt{PERMS}]\,E$ represents E running with permissions PERMS. A definition $u(x \triangleleft \texttt{CALL}).E$ introduces a function u; only callers with at least permissions CALL can invoke this function, supplying an argument for x. Multiple function definitions can be combined into a pool with the $\wedge$ operator. Function definitions, "let" and ν are binding operators for variables and names, respectively: the notions of free names *fn* and free variables *fv* arise as expected.

Table 2. Reduction semantics for π-Perms

(R-Call)

$$\frac{\texttt{CALL} \sqsubseteq \texttt{PERMS} \qquad \texttt{RECV} \sqsubseteq \texttt{PERMS}'}{n(x \triangleleft \texttt{CALL}).[\texttt{PERMS}']\,E \setminus [\texttt{PERMS}]\,\overline{n}\langle m \triangleright \texttt{RECV}\rangle \rightarrow [\texttt{PERMS}']\,E\{m/x\}}$$

(R-Return)

$$\mathsf{let}\ x = [\texttt{PERMS}]\,n\ \mathsf{in}\ E \rightarrow E\{n/x\}$$

(R-Context)

$$\frac{E \rightarrow E'}{\mathcal{C}[E] \rightarrow \mathcal{C}[E']}$$

(R-Struct)

$$\frac{E \Rrightarrow E_1 \rightarrow E_2 \Rrightarrow E'}{E \rightarrow E'}$$

Reduction contexts: $\mathcal{C}[\cdot] ::= \ \cdot \ \mid \ \mathsf{let}\ x = \mathcal{C}[\cdot]\ \mathsf{in}\ E \ \mid \ (\nu n)\,\mathcal{C}[\cdot] \ \mid \ D \setminus \mathcal{C}[\cdot]$

The formal semantics of π-Perms is given by the small-step reduction relation $E \rightarrow E'$ defined in Table 2.

Rule (R-Call) implements the security "cross-check" between caller and callee, which we discussed earlier: if either the caller is not assigned permissions CALL, or the callee is not granted permissions RECV, then the invocation fails. Whenever the invocation is successful, the expression runs with the permissions of the callee. The other rules are essentially standard, we just note that (R-Struct) closes reduction under *heating*, an asymmetric variant of the standard structural congruence relation. The heating relation $E \Rrightarrow E'$ allows to syntactically rearrange E into E', for instance by exchanging the order of the function definitions and by extruding the scope of bound names (see the online technical report for a complete definition of the heating relation).

3.2 π-Perms vs Android

Intents. π-Perms can encode both implicit and explicit intents. Communication in π-Perms is non-deterministic, in that a function invocation $\overline{n}\langle m \triangleright \texttt{RECV}\rangle$ can trigger any function definition $n(x \triangleleft \texttt{CALL}).E$ in the same scope, provided that the permission checks are satisfied. Technically, this non-determinism is achieved through the heating relation, which allows to liberally rearrange the pool of function definitions. Hence, communication in π-`Perms` naturally accounts for implicit intents, which represent the most interesting aspect of Android communication. Explicit intents can be recovered by univocally assigning each function definition with a distinct, unique permission: explicit communication is then encoded by requiring the callee to possess (at least) such permission.

Components. All of Android's intent-based component types are represented in π-`Perms` by means of function definitions. Activities in Android may be started by invoking the methods `startActivity` or `startActivityForResult`; in our calculus we treat the two cases uniformly, by having functions always return a result. Services may either be started by `startService` or become the end-point of a long-running connection with a client through an invocation to `bindService`. The former behaviour is modelled directly in π-`Perms` by a function call, while the latter is subtler and its encoding leads to some interesting findings (see below). Broadcast communication can be captured by a sequence of function invocations: this simple treatment suffices for our present security analysis.

Protection Mechanisms. π-`Perms` is defined around a generic complete lattice of permissions. In Android this lattice is built over permission sets, with set inclusion as the underlying partial order. The Android communication API only allows broadcast transmissions to be protected by permissions, namely requiring receivers to be granted specific permissions to get the intent. Function invocation in π-`Perms` accounts for the more general behaviour available to broadcast transmissions, since unprotected communication can be encoded simply by specifying $\bot$ as the permission required to the callee, as in $\overline{n}\langle m \triangleright \bot\rangle$.

Binders. In Android a component can invoke the method `bindService` to establish a connection with a service and retrieve an `IBinder` object, which transparently dispatches method calls from the client to the service. This behavior is captured in π-`Perms` by relying on its provision for dynamic component creation. To illustrate, let D contain the following service definition:

$$D \triangleq s(x \triangleleft \mathtt{C}).[\mathtt{P}]\,(\nu b)\,(b(y \triangleleft \bot).[\mathtt{P}]\,\overline{a}\langle y \triangleright \bot\rangle \setminus b) \qquad (1)$$

and consider the π-`Perms` encoding of a component binding to service s:

$$a(x \triangleleft \mathtt{P}).[\mathtt{P}]\,E \wedge D \setminus [\mathtt{C}]\,\mathsf{let}\ z = \overline{s}\langle n \triangleright \bot\rangle\ \mathsf{in}\ \overline{z}\langle n \triangleright \bot\rangle$$

Service s runs with permissions `P` and requires permissions `C` to establish a connection. When a connection is successfully established, the service returns a fresh binder b, encoded as a function granted the same permissions `P` as s; later, the client can perform an invocation to b (bound to z) to get access to the function a. The example unveils a potentially dangerous behaviour of the current Android implementation of `IBinder`'s: notice in particular that the function b may be invoked with no constraint, even though binding to s was protected by permissions `C`. We find this implementation potentially dangerous, since it is exposed to privilege escalation when binders are improperly disclosed.

Pending Intents. π-`Perms` can naturally encode the simple form of permission delegation enabled by pending intents: "by giving a `PendingIntent` to another application, you are granting it the right to perform the operation you have specified as if the other application was yourself (with the same permissions and identity)" [15]. This informal description perfectly fits the previous encoding of binders in π-`Perms`, in that any component exposed to the binder b is allowed to invoke the corresponding function and let it run with permissions `P`. Hence, pending intents can be modelled in the very same way as binders, and are exposed to the same weaknesses whenever they are inadvertently disclosed.

4 Privilege Escalation (Formally)

Davi *et al.* first pointed out a conceptual weakness in the Android permission system, showing that it is vulnerable to privilege escalation attacks [5]. To illustrate, consider three applications A, B and C. Application A is granted no permission; application B, instead, is granted permission `P`, which is needed to access C. Apparently, data and requests from A should not be able to reach C; on the other hand, if B can be freely accessed from A, then it may possibly act as a proxy between A and C.

We formalize a notion of safety against privilege escalation based on the IPC Inspection mechanism proposed by Felt *et al.* to dynamically prevent privilege escalation attacks on Android [11]. The idea behind IPC Inspection is simple: when an application receives a message from another application, a centralized reference monitor lowers the privileges of the recipient to the intersection of the privileges of the two interacting applications. A patched Android system implementing IPC Inspection is therefore protected against privilege escalation attacks "by design": we then take such a system as a reference specification and state a simulation-based notion of safety on top of it. As we discuss at the end of this section, the resulting definition provides an effective proof technique for the characterization of privilege escalation safety based on non-interference in [12].

To formalize the semantics of the IPC inspection mechanism, we first annotate each function definition of a given expression with a distinct label ℓ drawn from a denumerable set $\mathcal{L}$, disjoint from the set of values. The annotations make it possible to univocally identify the function triggered in response to each call, and hence trace the call chain. The IPC inspection semantics is then rendered formally by the labelled reduction relation $E \xrightarrow{\alpha}_i E'$ in Table 3, where α ranges uniformly over the set of annotation labels and the distinguished symbol $\cdot \notin \mathcal{L}$.

Note that, while the labelled transitions help tracking the dynamics of the call chains, the labels themselves do not have any import at runtime: in fact, function invocations do not mention labels at all and the semantics is still non-deterministic. We similarly label the original semantics in Table 2.

Let now $E_1 \asymp E_2$ denote two expressions that are syntactically equal but for their granted permissions (see the online technical report for a formal definition).

Definition 1 (IPC-Simulation). *A binary relation $\mathcal{R}$ contained in $\asymp$ is an* IPC-simulation *if and only if whenever $E_1 \mathcal{R} E_2$ and $E_1 \xrightarrow{\alpha} E_1'$ there exists E_2'*

Table 3. Reduction semantics for π-`Perms` under IPC Inspection

(R-Call-IPC)

$$\frac{\mathtt{RECV} \sqsubseteq \mathtt{PERMS}' \qquad \mathtt{CALL} \sqsubseteq \mathtt{PERMS}}{n^{\ell}(x \lhd \mathtt{CALL}).[\mathtt{PERMS}']\, E \setminus [\mathtt{PERMS}]\, \overline{n}\langle m \rhd \mathtt{RECV}\rangle \xrightarrow{\ell}_i [\mathtt{PERMS} \sqcap \mathtt{PERMS}']\, E\{m/x\}}$$

(R-Return-IPC)

$$\mathsf{let}\ x = [\mathtt{PERMS}]\, n\ \mathsf{in}\ E \xrightarrow{\cdot}_i E\{n/x\}$$

(R-Context-IPC)

$$\frac{E \xrightarrow{\alpha}_i E'}{\mathcal{C}[E] \xrightarrow{\alpha}_i \mathcal{C}[E']}$$

(R-Struct-IPC)

$$\frac{E \Rrightarrow E_1 \xrightarrow{\alpha}_i E_2 \Rrightarrow E'}{E \xrightarrow{\alpha}_i E'}$$

such that $E_2 \xrightarrow{\alpha}_i E_2'$ with $E_1' \mathcal{R} E_2'$. We say that E_1 is IPC-simulated *by E_2 (written $E_1 \preccurlyeq_{IPC} E_2$) iff there exists an IPC-simulation $\mathcal{R}$ such that $E_1 \mathcal{R} E_2$.*

The requirement $E_1 \asymp E_2$ guarantees that the labels that annotate the function definitions occurring in the two expressions are consistent (i.e., the same function bears the same label in E_1 and E_2) while disregarding any difference in the assigned permissions introduced upon reduction (cf. (R-Call) against (R-Call-IPC)). Given the previous definition, our notion of safety is immediate: an expression E is safe if and only if all its possible executions are oblivious to IPC Inspection being enabled or not.

Definition 2 (Safety). *An expression E is* safe *against privilege escalation if and only if $E \preccurlyeq_{IPC} E$.*

Though our definition is inspired by IPC Inspection, it reveals an important aspect which was never discussed before. Namely, we notice that improper disclosure of some specific data, such as binders or pending intents, may lead to the development of applications which are unsafe according to Definition 2. This is precisely the case of example (1) where b exercises permissions `P`, but can be disclosed to any component which is granted permissions `C`. A sample Android application suffering of a similar flaw is given in the online technical report.

Our notion of safety is already a strong property, but we target a more ambitious goal: we desire protection despite the best efforts of an active opponent. In our model an opponent is a malicious, but unprivileged, Android application installed on the same device. Notice that the term "unprivileged" is loosely used here: we are not assuming that the opponent is granted no permission at all, but rather that it is not assigned any sensitive permission beforehand (in that case, it would have no reason in escalating privileges). In a typical security analysis, one can single out all the permissions under the control of the opponent (e.g., `INTERNET`) and identify the set of these permissions with $\bot$.

Definition 3 (Opponent). *A definition O is an* opponent *if and only if each permission assignment in O is $\bot$.*

Definition 4 (Robust Safety). *An expression E is* robustly safe *against privilege escalation if and only if $O \setminus E$ is safe for all opponents O.*

Privilege Escalation and Non-interference. As we anticipated, a recent paper by Fragkaki *et al.* [12] proposes a definition of safety against privilege escalation inspired by the classic notion of non-interference for information flow control. Their definition essentially demands that any call chain ending in a "high" (permission-protected) component exists in a system only if it exists in a variant of same system, where the "low" (unprivileged) components have been pruned away. We can rephrase their notion in our setting and prove that our definition implies, and hence may be employed as a proof technique for, theirs.

Let $|E|_\ell$ denote the expression obtained from E by erasing all the function definitions labelled with $\ell' \neq \ell$ and which are granted permissions $\mathtt{P} \sqsubset \mathtt{CALL}$, where $\mathtt{CALL}$ are the permissions required to invoke the function identified by ℓ.

Definition 5 (NI-Safety). *An expression E is* NI-safe *if and only if, for every ℓ occurring in E and for every reduction sequence $E \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} E_n \xrightarrow{\ell} E_{n+1}$, there exist $E'_1, \dots, E'_{n+1}$ such that $|E|_\ell \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} E'_n \xrightarrow{\ell} E'_{n+1}$.*

Proposition 1 (Safety vs NI-safety). *Safety implies NI-safety.*

Proof. Let $E \preccurlyeq_{IPC} E$ and assume $E \xrightarrow{\alpha_1} E_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} E_n \xrightarrow{\ell} E_{n+1}$. Since $E \preccurlyeq_{IPC} E$, we know that $E \xrightarrow{\alpha_1}_i E'_1 \xrightarrow{\alpha_2}_i \dots \xrightarrow{\alpha_n}_i E'_n \xrightarrow{\ell}_i E'_{n+1}$ for some $E'_1, \dots, E'_{n+1}$ such that $E_1 \asymp E'_1, \dots, E_{n+1} \asymp E'_{n+1}$. By definition of the semantics $\xrightarrow{\alpha}_i$, we know that all the functions invoked in the call chain identified by $\alpha_1, \dots, \alpha_n$ must be granted at least the permissions $\mathtt{CALL}$ needed to invoke ℓ. Hence, such function definitions are present also in $|E|_\ell$ and we can mimic the very same trace there.

We can thus confirm that the IPC Inspection mechanism enforces a reasonable semantic security property and justify further our choice of taking it as the building block for our safety notion. With respect to NI-safety, our notion has the important advantage of enabling a powerful form of coinductive reasoning, which is central to proving our main result (Theorem 2 below).

A still open question is if the two notions of safety are actually equivalent. We notice that for non-deterministic transition systems (bi)simulation-based equivalences are typically finer than trace equivalences, but at the time of writing we were not able to identify a counterexample in our setting.

5 Preventing Privilege Escalation by Types and Effects

Types and Typing Environments. A type τ may be either Un or a function type $\mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}}$. Type Un is the base type, which is used both as a building block for function types and to encompass all the data which are under the control of the opponent. Types of the form $\mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}}$ are inhabited by functions which input arguments of type τ and return results of type τ'. Functions with this type can be invoked only by callers which are granted at least permissions $\mathtt{CALL}$, and should only be disclosed to components running with at least permissions $\mathtt{SECR}$. We define the *secrecy level* of a type τ, written $\mathcal{S}(\tau)$, as expected, by having $\mathcal{S}(\mathsf{Un}) = \bot$ and $\mathcal{S}(\mathsf{Fun}(\mathtt{CALL}, \tau \rightarrow \tau')^{\mathtt{SECR}}) = \mathtt{SECR}$. A typing *environment* Γ is a finite map from values to types. The *domain* of Γ, written $dom(\Gamma)$, is the set of the values on which Γ is defined.

Typing Values. The typing rules for values are simple and given in Table 4.

Table 4. Typing rules for values

$$\text{(T-Proj)}\ \frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau} \qquad \text{(T-Pub)}\ \frac{\Gamma \vdash v : \tau \qquad \mathcal{S}(\tau) = \bot}{\Gamma \vdash v : \mathsf{Un}}$$

Rule (T-Proj) is standard, while rule (T-Pub) makes it possible to treat all public data as “untyped”, since they may possibly be disclosed to the opponent.

Typing Expressions. The typing rules for expressions are in Table 5. The main judgement $\Gamma \vdash_{\mathtt{P}} E : \tau \blacktriangleright \mathtt{Q}$ is read as: expression E, running with permissions $\mathtt{P}$, has type τ in Γ and exercises at most permissions $\mathtt{Q}$ throughout its execution. We also define an auxiliary judgement $\Gamma \vdash D$ to be read as: definition D is well-formed in Γ. The two judgement forms are mutually dependent.

We first notice that our effect system discriminates between *granted* and *exercised* permissions. For instance, the expression $a(x \triangleleft \bot).[\mathtt{P}]\,\overline{b}\langle n \triangleright \bot\rangle \setminus E$ could either be well-typed or not, even though the function a is publicly known, but is granted permissions $\mathtt{P} \sqsupset \bot$. The crux here is if the permissions $\mathtt{P}$ must be actually exercised or not to perform the invocation to b.

Apparently, we could enforce protection against privilege escalation by simply checking for each function definition that the privileges exercised by the function body are at most equal to the privileges required to invoke the function. However, since binders and pending intents allow indiscriminate access to potentially privileged components, our type system must also assign an appropriate secrecy level to these sensitive data and prevent their inadvertent disclosure. It turns out that in rule (T-Def) we must actually check that the permissions $\mathtt{Q}$ exercised by the function body must be at most equal to the join between the permissions $\mathtt{CALL}$, needed to pass the security runtime checks upon invocation, and the permissions $\mathtt{SECR}$, needed to learn the name of the function.

Interestingly, the opponent can play an active role in trying to get binders and pending intents under its control. In particular, by using rules (T-Def-Un) and (T-Call-Un), it can define arbitrary new functions and invoke existing ones, completely disregarding the restrictions enforced by typing. Protecting well-typed components requires then some care: for instance, in rule (T-Def) we must type-check public functions under the additional assumption that their input parameter is provided by the opponent with type Un; of course, in this case

Table 5. Typing rules for definitions and expressions

(T-Def)
$$\frac{\Gamma \vdash u : \mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}} \quad \Gamma, x : \tau \vdash_{\top} E : \tau' \blacktriangleright \mathtt{Q} \quad \mathtt{Q} \sqsubseteq \mathtt{CALL} \sqcup \mathtt{SECR} \quad \mathtt{CALL} \sqcup \mathtt{SECR} = \bot \Rightarrow \Gamma, x : \mathsf{Un} \vdash_{\top} E : \mathsf{Un} \blacktriangleright \bot \quad x \notin dom(\Gamma)}{\Gamma \vdash u(x \lhd \mathtt{CALL}).E}$$

(T-Conj)
$$\frac{\Gamma \vdash D_1 \quad \Gamma \vdash D_2}{\Gamma \vdash D_1 \wedge D_2}$$

(T-Eval)
$$\frac{\Gamma \vdash D \quad \Gamma \vdash_{\mathtt{P}} E : \tau \blacktriangleright \mathtt{Q}}{\Gamma \vdash_{\mathtt{P}} D \setminus E : \tau \blacktriangleright \mathtt{Q}}$$

(T-Call)
$$\frac{\Gamma \vdash u : \mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}} \quad \Gamma \vdash v : \tau \quad \bot \sqsubset \mathtt{RECV} \sqcup \mathtt{SECR} \quad \mathtt{CALL} \sqcup \mathtt{SECR} \sqsubseteq \mathtt{P}}{\Gamma \vdash_{\mathtt{P}} \overline{u}\langle v \rhd \mathtt{RECV}\rangle : \tau' \blacktriangleright \mathtt{CALL} \sqcup \mathtt{SECR}}$$

(T-Val)
$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash_{\mathtt{P}} v : \tau \blacktriangleright \mathcal{S}(\tau)}$$

(T-Fail)
$$\frac{\Gamma \vdash u : \mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}} \quad \Gamma \vdash v : \tau'' \quad \mathtt{RECV} \sqcup \mathtt{SECR} = \bot \Rightarrow \mathcal{S}(\tau'') = \bot \quad \mathtt{CALL} \not\sqsubseteq \mathtt{P}}{\Gamma \vdash_{\mathtt{P}} \overline{u}\langle v \rhd \mathtt{RECV}\rangle : \mathsf{Un} \blacktriangleright \mathtt{P}}$$

(T-Perms)
$$\frac{\Gamma \vdash_{\mathtt{Q}} E : \tau \blacktriangleright \mathtt{R} \quad \mathtt{Q} \sqsubseteq \mathtt{P}}{\Gamma \vdash_{\mathtt{P}} [\mathtt{Q}]\, E : \tau \blacktriangleright \mathtt{R}}$$

(T-Let)
$$\frac{\Gamma \vdash_{\mathtt{P}} E : \tau \blacktriangleright \mathtt{Q} \quad \Gamma, x : \tau \vdash_{\mathtt{P}} E' : \tau' \blacktriangleright \mathtt{R} \quad x \notin dom(\Gamma)}{\Gamma \vdash_{\mathtt{P}} \mathsf{let}\ x = E\ \mathsf{in}\ E' : \tau' \blacktriangleright \mathtt{Q} \sqcup \mathtt{R}}$$

(T-Restr)
$$\frac{\Gamma, n : \tau \vdash_{\mathtt{P}} E : \tau' \blacktriangleright \mathtt{Q} \quad n \notin dom(\Gamma)}{\Gamma \vdash_{\mathtt{P}} (\nu n)\, E : \tau' \blacktriangleright \mathtt{Q}}$$

(T-Def-Un)
$$\frac{\Gamma \vdash u : \mathsf{Un} \quad \Gamma, x : \mathsf{Un} \vdash_{\bot} E : \mathsf{Un} \blacktriangleright \bot \quad x \notin dom(\Gamma)}{\Gamma \vdash u(x \lhd \mathtt{CALL}).E}$$

(T-Call-Un)
$$\frac{\Gamma \vdash u : \mathsf{Un} \quad \Gamma \vdash v : \mathsf{Un}}{\Gamma \vdash_{\bot} \overline{u}\langle v \rhd \mathtt{RECV}\rangle : \mathsf{Un} \blacktriangleright \bot}$$

no privilege must be exercised. Similarly, in rule (T-Call) we cannot trust the return type of a function when the invocation can be dispatched to the opponent: this justifies the third premise of the rule.

Rule (T-Fail) allows to provide an argument of arbitrary type to any function which will never be invoked at runtime, since the caller is granted permissions P, but the function requires permissions CALL $\not\sqsubseteq$ P to be invoked. Again, the information CALL in the function type can be trusted only when the function is not defined by the opponent, hence some additional care is needed to prevent secrecy violations in that case (see the third premise of the rule). Note that, due to such a possible interaction with the opponent, the exercised permissions are conservatively assumed to be P, i.e., all the permissions granted to the caller.

We conclude the description of the type system with an important remark on expressiveness. Some of the constraints imposed by our typing rules are rather restrictive for practical use, but are central to enforcing the conditions of

Definition 2 and its robust variant. Our implementation, however, features a number of escape hatches based on Java annotations to keep programming practical, much in the spirit of the declassification/endorsement constructs customary to the literature on information-flow control. We discuss this point further in Section 6.

Example Type-Checking. We briefly discuss how example (1) is deemed as ill-typed according to our type discipline. We first note that, since function a requires permissions $\mathtt{P}$ to be called, the invocation $\overline{a}\langle y \triangleright \bot\rangle$ is assigned at least the effect $\mathtt{P}$ by (T-Call). Hence, the only possible way to type-check the function definition $b(y \triangleleft \bot).[\mathtt{P}]\,\overline{a}\langle y \triangleright \bot\rangle$ through (T-Def) is by assigning b a function type τ such that $\mathcal{S}(\tau) = \mathtt{P}$. Assuming that the service s is a public component, this implies that the function definition $s(x \triangleleft \mathtt{C}).[\mathtt{P}]\,(\nu b)\,\ldots \setminus b$ is ill-typed by (T-Def), since the effect $\mathtt{P}$ assigned to the service body b by (T-Val) is not lesser or equal to the permissions $\mathtt{C}$ required to invoke the service s.

Formal Results. The safety result below follows by a "simulation-aware" variant of a standard Subject Reduction theorem for our type system, which captures the step-by-step relationships between the standard semantics and our reference semantics. The proof relies on a co-inductive argument enabled by the Subject Reduction theorem: full details can be found in the online technical report.

Theorem 1 (Type Safety). *If $\Gamma \vdash_{\top} E : \tau \blacktriangleright \mathtt{P}$ for any $\mathtt{P}$, then $E \preccurlyeq_{IPC} E$.*

The next result states that our type system does not constrain the opponent. Its proof follows by a simple structural induction.

Lemma 1 (Opponent Typability). *Let O be an opponent and let $\Gamma \vdash u : \mathsf{Un}$ for all $u \in fnfv(O)$, then $\Gamma \vdash O$.*

By combining the two previous results, we can prove our main theorem.

Theorem 2 (Robust Safety). *Let $\mathcal{S}(\tau) = \bot$ for every u such that $\Gamma(u) = \tau$. If $\Gamma \vdash_{\top} E : \tau \blacktriangleright \mathtt{P}$ for any $\mathtt{P}$, then E is robustly safe against privilege escalation.*

6 Implementation

We have implemented the type system as a tool (`Lintent`) designed as a plug-in for Android `Lint`, the widely popular utility distributed with Android's ADT.

`Lintent` performs a number of static checks over permissions usage, analyzing the application source code and the manifest permission declarations, and eventually warning the developer in case of potential attack surfaces for privilege escalation scenarios. As a byproduct of its analysis, `Lintent` is able to detect over-privileged or under-privileged applications, and suggest fixes. Additionally, `Lintent` infers and records the types of data injected into and extracted from intents, while tracking the flow of inter-component message passing. This is needed to prevent privilege escalation attacks exploiting improper disclosure of binders or pending intents, and at the same time proves very effective in detecting common programming errors related to misuse of intents [16].

`Lintent` analyzes Java source code: in principle, the same analysis could be performed on the Java bytecode, though reasoning about types at the bytecode level is arguably more demanding than at source level [14]. Below, we give a brief overview of the main features of the tool and of the the main challenges we had to face during its development.

Type Reconstruction. The hardest challenge for the implementation is related to the widespread use of "untyped" coding patterns supported by the current Android API. Consider, for instance, a simple scenario of intent usage with multiple data types:

```
class SenderActivity extends Activity {
   static class MySer implements Serializable { ... }

   void mySenderMethod() {
      Intent i = new Intent(this, ReceiverActivity.class);
      i.putExtra("k1", 3);
      i.putExtra("k2", "some_string");
      i.putExtra("k3", new MySer());
      startActivityForResult(i,0);
   }
}
```

On the recipient side, intent "extras" are retrieved by freely accessing the intent as if it was a dictionary, so the receiver may actually retrieve data of unexpected type and fail at runtime, or disregard altogether some keys provided by the sender [16].

```
class ReceiverActivity extends Activity {
    static class WS implements Serializable { ... }

    void onCreate(Bundle savedInstanceState) {
        Intent i = getIntent();
        String k1 = i.getStringExtra("k1"); // run-time type error!
        WS o = (WS)i.getSerializableExtra("k3"); // dynamic cast fails!
        // data associated to k2 is never extracted!
    }
}
```

The example highlights a total lack of static control over standard intents manipulation operations: with these premises, no type-based analysis can be soundly performed. For this reason, intents are treated in `Lintent` as record types of the form $\{k_1 : T_1, \ldots, k_n : T_n\}$, where each k_i is a string constant and each T_i is a Java type. This enforces a much stronger discipline on data passing between components, which is consistent with our type system, where a function type $\mathsf{Fun}(\mathtt{CALL}, \tau \to \tau')^{\mathtt{SECR}}$ constrains the caller in providing an argument of type τ

and the callee in returning a result of type τ'. A similar discipline is crucial in Android applications to protect the secrecy of binders and pending intents.

Notice that, since the `putExtra` method is overloaded to different types, the type of the second argument of each call must be reconstructed in order to keep track of the actual type of the value bound to each key. As a valuable byproduct of this analysis, `Lintent` is able to warn the user in case of intents misuse.

Partial Evaluation. As noted above, each piece of data put into an intent must be bound to a key, hence an intent object can be seen as a dictionary of the form $\{k_1 \mapsto v_1, \ldots, k_n \mapsto v_n\}$. Unfortunately, the dictionary keys are run-time (String) objects and therefore plain expressions in Java. Whether they happen to be string literals or the result of complex method calls computing a String object is irrelevant: in any case they belong to the run-time world. The very same problem arises for result codes and Intent constructor invocations: both the sender component and the recipient class object supplied as arguments could be results of computations, and the same holds true for action strings in case of implicit intent construction. Partial evaluation is required for reconstructing the intent record type labels described above.

API Signatures and Permissions. Implementing the rules of the type system for π-`Perms` requires a preliminary analysis to detect the corresponding patterns in the Android source code. The analysis is far from trivial given the complexity of the Android communication API, which offers several different patterns to implement inter-component communication. Moreover, many Android API calls require non-empty permission sets and must be detected and tracked by our tool: `Lintent` retrieves a set of mappings between API method signatures and permissions from a set of external files[2], which are thus updatable with no need to rebuild the tool. Finally, `Lintent` must perform type resolution for third-party libraries: access to `jar` files must be granted to the tool to let it inspect the contents of imported packages and classes through the `javap` disassembler.

Java Annotations Support. We rely on Java annotations to provide some escape hatches from the tight discipline imposed by `Lintent`. Several privileged components intentionally expose functionalities, thus we define annotations of the form `@priv{endorse="P"}` to mark methods such as `onCreate()` with a set of permissions `P` that the type-checker will disregard. More precisely, if the method exercises the permissions set `Q`, the associated component is deemed well-typed as long as it is protected with at least permissions `Q`\\`P`. A similar treatment is implemented for pending intents based on the annotation `@priv{declassify="P"}`, to lower the secrecy level of such objects computed by `Lintent`.

7 `Lintent`: Typing Experiments and Findings

At the time of writing `Lintent` is able to type-check activities, started services and broadcast receivers. The current prototype should be considered in alpha

[2] Currently such permission map files are those distributed with Stowaway [10].

stage, as we are currently performing tests, fixing bugs and adding support for some missing Java language features. Still, we were able to analyze some existing open-source applications from the Google Play store and identify previously unknown privilege escalation attacks on them. In our case studies we performed a code refactoring to avoid the usage of some Java features which are still unsupported by `Lintent`, like reflective calls and nested classes. However, our findings are confirmed by running the original applications on a Nexus device.

The first case study we consider is `APN-Switch`, a widget that allows to enable and disable the device data connection with a click. Of course, these network operations are sensitive, hence the application requires the permission `CHANGE_NETWORK_STATE` to be installed. Unfortunately, `APN-Switch` is exposed to privilege escalation attacks: an unprivileged malicious application can forge an intent to the action string `ch.blinkenlights.android.apnswitch.CLICK` and simulate a click of the user on the widget, thus enabling (or disabling) the device data connection as if it were granted the `CHANGE_NETWORK_STATE` permission.

Our second case study is `Wifi Fixer`, a small application aimed at fixing several problems with the Android wifi. Also `Wifi Fixer` suffers of privilege escalation attacks, since it requires the permission `CHANGE_WIFI_STATE` to toggle on and off the wifi connection, but any unprivileged application can send an intent to the action string `org.wahtod.wififixer.ACTION_WIFI_OFF` to disconnect the wifi. Interestingly, the widget handling the wifi connection is declared as an internal component, hence it cannot receive intents from third-party applications; however, a public broadcast receiver in the application can act as a proxy to the widget, thus allowing to escalate privileges.

Both `APN-Switch` and `Wifi Fixer` are released on the official Google Play store, hence available to a wide audience. We argue that `Lintent` can prove helpful not only in detecting malicious code lying within existing source programs, but also in assisting well-meaning developers in identifying potential attack surfaces for privilege escalation and many other common programming mistakes, way before their applications reach the Google Play store.

8 Related Work

The literature on Android application security is substantial, as reported in a recent survey by Enck [6].

Android Permissions. Davi *et al.* [5] were the first to point out the weaknesses of the Android permission system with respect to privilege escalation attacks. Later, Felt *et al.* proposed IPC Inspection as a possible runtime protection mechanism [11]. Though effective, IPC Inspection may induce substantial performance overhead, as it requires to keep track of different application instances to make the protection mechanism precise. In a recent paper, Bugiel *et al.* describe a sophisticated runtime framework for enforcing protection against privilege escalation attacks [2]. Notably, their solution comprises countermeasures against colluding applications, an aspect which is neglected by both IPC Inspection and

`Lintent`. Providing such guarantees, however, requires a centralized solution built over the operating system. Our approach is complementary: runtime protection is useful against malicious applications which reach the Android market, while static analysis techniques can prove helpful for well-meaning developers who wish to assess the robustness of their applications. Finally, Felt *et al.* proposed Stowaway, a tool for detecting overprivilege in Android applications [10]. In our implementation we take advantage of their permission map, which relates API method calls to their required permissions.

Android Communication. Chin *et al.* [4] were the first to study the threats related to the Android message-passing system. They provide also a tool, ComDroid, which is able to detect potential vulnerabilities in the usage of intents. ComDroid does not provide any formal guarantee about the effectiveness of the proposed secure communication guidelines; in our work, instead, we reason about intents usage in a formal calculus and we are able to confirm many previous observations as sound programming practices. ComDroid does not address the problem of detecting privilege escalation attacks. The robustness of inter-component communication in Android has been studied also by Maji *et al.* through fuzzy testing techniques, exposing some interesting findings [16]. Their empirical methodology, however, does not provide any clear understanding of the correct programming patterns for communication.

Formal Models. π-`Perms` is partly inspired by a core formal language proposed by Chaudhuri [3]. With respect to Chauduri's model, π-`Perms` provides a more thorough treatment of the Android system, including implicit communication, runtime registration of new components, service binding and pending intents. In later work, Fuchs *et al.* build on the calculus proposed by Chaudhuri to implement SCanDroid, a provably sound static checker of information-flow properties of Android applications [13]. Another work by Fragkaki *et al.* discusses a number of enhancements over the Android permission system and validates their effectiveness in an abstract model [12] (cf. Section 4). The focus of the work remains on runtime protection mechanisms, however, as opposed to static analysis. The paper also discusses some issues related to controlled delegation, but it does it independently from privilege escalation. Finally, Armando *et al.* proposed a formal model of the Android operating system and a verification technique based on history expressions [1]. However, any specific security analysis is left for future work and no implementation is provided.

9 Conclusions

We have proposed a sound type-based analysis technique targeted at the static detection of privilege escalation attacks on Android, and developed `Lintent`, a prototype security type-checker which implements our analysis. Our tool addresses a number of engineering challenges which are central to the practical development of any sound type-checker for Android applications. We showed the effectiveness of our tool by unveiling real attacks on existing applications.

As part of our future work, we want to focus on the study of robust declassification and endorsement programming patterns in our formal framework, to assess the impact on security of the Java annotations discussed in Section 6. On the practical side, we want to further develop `Lintent` and add support for many features of the Android platform which are still missing. We also plan to integrate `Lintent` with a frontend to a decompiler as `ded` [8] to support the analysis of third-party applications.

Acknowledgements. Work partially supported by MIUR PRIN Project "CINA: Compositionality, Interaction, Negotiation and Autonomicity", and conducted in cooperation with SMC Treviso s.r.l. The third author was supported by a EU-Regione Veneto funded fellowship within the POR FESR 2007 – 2013 Program, Action 1.1.3.

References

1. Armando, A., Costa, G., Merlo, A.: Formal modeling and verification of the Android security framework. In: TGC (2012)
2. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B.: Towards taming privilege-escalation attacks on Android. In: NDSS (2012)
3. Chaudhuri, A.: Language-based security on Android. In: PLAS, pp. 1–7 (2009)
4. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: MobiSys, pp. 239–252 (2011)
5. Davi, L., Dmitrienko, A., Sadeghi, A.-R., Winandy, M.: Privilege escalation attacks on android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 346–360. Springer, Heidelberg (2011)
6. Enck, W.: Defending users against smartphone apps: Techniques and future directions. In: Jajodia, S., Mazumdar, C. (eds.) ICISS 2011. LNCS, vol. 7093, pp. 49–70. Springer, Heidelberg (2011)
7. Enck, W., Gilbert, P., gon Chun, B., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: OSDI, pp. 393–407 (2010)
8. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of Android application security. In: USENIX Security Symposium (2011)
9. Enck, W., Ongtang, M., McDaniel, P.D.: Understanding Android security. IEEE Security & Privacy 7(1), 50–57 (2009)
10. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: CCS, pp. 627–638 (2011), `http://www.android-permissions.org/`
11. Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission re-delegation: Attacks and defenses. In: USENIX Security Symposium (2011)
12. Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and enhancing Android's permission system. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 1–18. Springer, Heidelberg (2012)
13. Fuchs, A.P., Chaudhuri, A., Foster, J.S.: Scandroid: Automated security certification of Android applications, Technical report, University of Maryland (2009)
14. Gagnon, E.M., Hendren, L., Marceau, G.: Efficient inference of static types for java bytecode. In: SAS 2000. LNCS, vol. 1824, pp. 199–220. Springer, Heidelberg (2000)
15. Google Inc: Reference documentation for android.app.PendingIntent, `http://developer.android.com/reference/android/app/PendingIntent.html`
16. Maji, A.K., Arshad, F.A., Bagchi, S., Rellermeyer, J.S.: An empirical study of the robustness of inter-component communication in Android. In: DSN (2012)

Honesty by Typing

Massimo Bartoletti[1], Alceste Scalas[1], Emilio Tuosto[2], and Roberto Zunino[3]

[1] Università degli Studi di Cagliari, Italy
{bart,alceste.scalas}@unica.it
[2] University of Leicester, UK
emilio@mcs.le.ac.uk
[3] Università degli Studi di Trento and COSBI, Italy
roberto.zunino@unitn.it

Abstract. We propose a type system for a calculus of contracting processes. Processes may stipulate contracts, and then either behave honestly, by keeping the promises made, or not. Type safety guarantees that a typeable process is *honest* — that is, the process abides by the contract it has stipulated in all possible contexts, even those containing dishonest adversaries.

1 Introduction

Most approaches to the formal specification of concurrent systems typically assume that components behave *honestly*, in that they always adhere to some agreed specification (e.g., a behavioural type), under the assumption that the static behaviour safely over-approximates the dynamic one. We argue that this assumption is unrealistic in scenarios where competition prevails against cooperation. Indeed, in a competitive scenario components may act selfishly, and diverge from the agreed specification.

We envision a *contract-oriented computing* paradigm [2], for the design of distributed components which use *contracts* to discipline their interaction. In CO_2 [2], a process may advertise contracts; a session is established processes advertising *compliant* contracts, similarly to other session-centric calculi. This session dictates the actions needed to realise their contracts to processes. A distinguished feature of CO_2 is that processes are not supposed to respect their contracts, nor are they bound to them by an enforcing mechanism.

More realistically, *dishonest* processes may avoid to perform some actions they have promised in their contracts. This may happen either intentionally, e.g. a malicious process trying to swindle the system, or unintentionally, e.g. because of some implementation bug (possibly exploited by some adversary). In both cases, the infrastructure can determine which process has caused the violation, and adequately punish it. A crucial problem is how to guarantee that a process will behave honestly, in *all possible contexts*. If such guarantee can be given, the process is protected both against unintentional bugs, and against (apparently honest) adversaries which try to make it sanctioned.

A negative result in [3] is that determining if a process is honest is an undecidable problem for a relevant class of contracts (introduced in [10], and refined in [11], for modelling WSDL and WSCL contracts). The problem is how to find a computable approximation of honesty, which implies the dynamic one.

Example. Let us consider an on-line store (participant A), which sells apples (a) and bottles of an expensive italian Brunello wine (b). Selling apples is quite easy: once an order is placed, A accepts it (with the feedback $\overline{\mathsf{ok}}$) and waits for a payment (pay) before shipping the goods ($\overline{\mathsf{ship\text{-}a}}$). However, if expensive bottles of Brunello are ordered, the store is entitled to either decline the order (by answering $\overline{\mathsf{no}}$), or accept it (and, as above, ship the item after the payment). Using external ($+$) and internal ($\oplus$) choice, the store contract can be modelled as

$$c \;=\; \mathsf{a}.\overline{\mathsf{ok}}.\,\mathsf{pay}.\overline{\mathsf{ship\text{-}a}} \;\;+\;\; \mathsf{b}.\left(\overline{\mathsf{no}} \;\oplus\; \overline{\mathsf{ok}}.\mathsf{pay}.\overline{\mathsf{ship\text{-}b}}\right)$$

External choice requires the other party to decide how to drive the contract evolution; in this case, the customer chooses between apples a and bottles b. Internal choice, instead, allows the advertising party to choose; in this case, the store selects either $\overline{\mathsf{ok}}$ or $\overline{\mathsf{no}}$.

Intuitively, contract compliance hinges on the duality between internal and external choices. Hence, to sell its goods, the store needs to find an agreement with a participant advertising a contract *compliant* with its contract c, such as:

$$d \;=\; \overline{\mathsf{b}}\,.\,(\mathsf{ok}.\overline{\mathsf{pay}}.\mathsf{ship\text{-}b} \;+\; \mathsf{no})$$

A buyer B advertising d wants to buy Brunello: she promises to select $\overline{\mathsf{b}}$ (dual of b in the store contract), and then offers an external choice that lets the store choose between ok or no; in the first case, she promises to pay and wait for shipment.

In CO$_2$, the behaviour of each participant is a *process* capable of advertising contracts and executing the actions required to honour them. For instance, the store A can advertise its contract c by firing the prefix $\mathsf{tell}_{\mathsf{K}} \downarrow_x c$, where the index x in $\downarrow_x c$ is the name of a *channel* of A, and K is the name of an external broker, whom the contract is being advertised to. We shall not specify the behaviour of K, and just assume that when it finds a contract compliant with c, a session is established between A and another participant, say B. Technically, x is replaced with a fresh session name s. Participants A and B will then use such session to perform the actions required by their contracts.

A possible specification of the store A is e.g.:

$$\begin{aligned} P_M \;&=\; (x)\,(\mathsf{tell}_{\mathsf{K}}{\downarrow_x}\, c\,.\,(\mathsf{do}_x\,\mathsf{a}\,.\,X_M(x) \;+\; \mathsf{do}_x\,\mathsf{b}\,.\,X_M(x))) \\ X_M(x) \;&\stackrel{\text{def}}{=}\; \mathsf{do}_x\,\overline{\mathsf{ok}}\,.\,\mathsf{do}_x\,\mathsf{pay}\,.\,\mathsf{ask}_x\,\overline{\mathsf{ship\text{-}a}}?\,.\,\mathsf{do}_x\,\overline{\mathsf{ship\text{-}a}} \end{aligned}$$

Here, A creates a private channel x, and advertises the contract c. Once a session starts, P_M can accept an order for a or b on x. This is modelled by the guards $\mathsf{do}_x\,\mathsf{a}$ and $\mathsf{do}_x\,\mathsf{b}$ of the choice operator $+$ (not to be confused with $+$ of contracts). In both cases, the process $X_M(x)$ is invoked. There, A accepts the transaction with ok, and waits for payment. Then A checks whether the contract requires to ship apples: if the query $\mathsf{ask}_x\,\overline{\mathsf{ship\text{-}a}}?$ passes, the goods are shipped. Otherwise, when the customer B orders Brunello, A maliciously gets stuck, and so B has paid for nothing. This store is *dishonest*: it does not respect its own contract c.

Consider now a non-malicious implementation of the store. Before accepting orders, the store requires an insurance to cover shipment damages. With the contract $c_p = \overline{\mathsf{payP}} \,.\, (\overline{\mathsf{cover}} \oplus \overline{\mathsf{cancel}})$, A promises to pay $(\overline{\mathsf{payP}})$ and then choose between getting the coverage, or cancelling the request. The new store process is:

$$\begin{aligned}
P_N \;&=\; (x,y)\big(\mathsf{tell}_{\mathsf{C}} \downarrow_y c_p \,.\, \mathsf{do}_y\, \overline{\mathsf{payP}} \,.\, \mathsf{tell}_{\mathsf{K}} \downarrow_x c\,. \\
&\qquad\qquad \big(\mathsf{do}_x\, \mathsf{a} \,.\, \mathsf{do}_x\, \overline{\mathsf{ok}} \,.\, X_N(x) + \mathsf{do}_x\, \mathsf{b} \,.\, Y_N(x,y)\big)\big) \\
X_N(x) \;&\stackrel{\text{def}}{=}\; \mathsf{do}_x\, \mathsf{pay} \,.\, \big(\mathsf{ask}_x\, \overline{\mathsf{ship\text{-}a}}? \,.\, \mathsf{do}_x\, \overline{\mathsf{ship\text{-}a}} + \mathsf{ask}_x\, \overline{\mathsf{ship\text{-}b}}? \,.\, \mathsf{do}_x\, \overline{\mathsf{ship\text{-}b}}\big) \\
Y_N(x,y) \;&\stackrel{\text{def}}{=}\; \mathsf{do}_y\, \overline{\mathsf{cover}} \,.\, \big(\mathsf{do}_x\, \overline{\mathsf{ok}} \,.\, X_N(x) \;+\; \tau \,.\, \mathsf{do}_x\, \overline{\mathsf{no}}\big)
\end{aligned}$$

The store A first requests an insurance by advertising c_p to an insurance company (say C); once C agrees, A pays the premium (on channel y), and then advertises c; once an agreement with a customer B is reached, A waits for a or b orders. If apples are requested, A acknowledges $(\overline{\mathsf{ok}})$ and invokes $X_N(x)$; there, A waits for payment, checks which good has to be shipped, and actually ships it. Otherwise, if Brunello is requested, $Y_N(x,y)$ is invoked: there, A requests the insurance coverage paid in advance; then, either the order is accepted and $X_N(x)$ is invoked for payment and shipment (as above), or the transaction is declined after an internal action τ (e.g. a wake up after a timeout).

This implementation is not malicious as the first attempt: it is keen to ship the goods, but it is not honest either due to the interaction between A and the insurance company. If C does not execute cover, A gets stuck on $\mathsf{do}_y\, \overline{\mathsf{cover}}$, unable to honour c by providing the expected $\overline{\mathsf{ok}}/\overline{\mathsf{no}}$. Furthermore, A is dishonest w.r.t. c_p: the premium is paid in advance, but A may never perform $\mathsf{do}_y\, \overline{\mathsf{cover}}$ nor $\mathsf{do}_y\, \overline{\mathsf{cancel}}$ — e.g. if no agreement on c is found, or if the customer B is stuck, or if B simply chooses to buy apples. Thus, due to implementation naïveties, A may be blamed due to the unexpected (or malicious) behaviour of other participants.

Contributions. Our main contribution is a type discipline for statically ensuring when a CO$_2$ process is *honest*. The need for a static approximation is due to the fact that honesty is an undecidable property [3]. Our type system associates behavioural types to process channels. Checking honesty on these abstractions is decidable (Th. 1). We establish subject reduction (Th. 2) and progress (Th. 3), which are then used to prove type safety: typeable processes are honest (Th. 4).

Due to space constraints, we publish the proofs and further examples in a separate report [1].

2 A Calculus of Contracting Processes

The calculus CO$_2$ is parametric on the contract model, i.e. on the language of contracts. Contract models for CO$_2$ have been defined based e.g. on formulae of Propositional Contract Logic [4], and on CCS processes [2]. In this paper we focus on the two-party contracts of [11], as in [3]. Due to space limits, here we only give the main definitions, and refer the reader to [1] for the full details.

We assume a set of participants $\mathsf{A}, \mathsf{B}, \ldots$, and a set of *atoms* $\mathsf{a}, \mathsf{b}, \ldots$, that represent the actions performed by participants. We use an involution $\bar{\mathsf{a}}$, as in CCS, and assume a distinguished atom e, modelling a successfully terminated participant, such that $\mathsf{e} = \bar{\mathsf{e}}$.

A *unilateral contract* of [11] is a CCS-like process c which models the promised behaviour of a single participant. An internal sum $c = \bigoplus_{i \in \mathcal{I}} \mathsf{a}_i \,;\, c_i$ requires a participant (say, A) to choose one of the actions a_i, do it, and then behave according to its continuation c_i. An external sum $c = \sum_{i \in \mathcal{I}} \mathsf{a}_i \,.\, c_i$ requires A to offer a choice among all the branches; if a_i is chosen by the other participant, then A must continue according to c_i. Finally, $c = \mathit{ready}\ \mathsf{a}.c'$ models an obligation of A to do a, and then to proceed as c'. We denote (guarded) recursive contracts with $\mathit{rec}\ X \,.\, c$, and we let $E = \mathit{rec}\ X \,.\, \mathsf{e} \,;\, X$.

A *bilateral contract* $\gamma = \mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d$ combines the contracts of two participants. Its behaviour is given in [3] as a LTS. Its main rules regulate the interaction between the internal choices of A, and the external choices of B:

$$\mathsf{A}\ \mathit{says}\ (\bar{\mathsf{a}} \,;\, c \oplus c') \mid \mathsf{B}\ \mathit{says}\ (\mathsf{a} \,.\, d + d') \xrightarrow{\mathsf{A}\ \mathit{says}\ \mathsf{a}} \mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ \mathit{ready}\ \mathsf{a}.d \quad (1)$$

$$\mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ \mathit{ready}\ \mathsf{a}.\ d \xrightarrow{\mathsf{B}\ \mathit{says}\ \mathsf{a}} \mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d \quad (2)$$

By (1), if A chooses branch $\bar{\mathsf{a}}$ in her internal sum, then B is committed to the branch a in his external sum. We mark the selected branch with *ready* a, and discard the others. This enables (2) where B takes the marked branch. When an internal choice is not matched by an action in the external choice of the partner, the contract gets stuck. Intuitively, two contracts c, d are *compliant*, $c \bowtie d$ in symbols, when this situation cannot occur. The formal definition of compliance builds upon that of *ready sets*. For a contract c, the ready sets $\mathit{RS}(c)$ are:

$$\begin{array}{llll} \{\{\mathit{ready}\ \mathsf{a}\}\}, & \text{if } c = \mathit{ready}\ \mathsf{a}.c' & \mathit{RS}(c'), & \text{if } c = \mathit{rec}\ X \,.\, c' \\ \{\{\mathsf{a}_i\} \mid i \in I\}, & \text{if } c = \bigoplus_{i \in I} \mathsf{a}_i \,;\, c_i \text{ and } I \neq \emptyset & \{\{\mathsf{a}_i \mid i \in I\}\}, & \textit{if } c = \sum_{i \in I} \mathsf{a}_i \,.\, c_i \end{array}$$

Then, the relation $\bowtie$ on contracts is the largest relation preserved by contract transitions such that, whenever $c \bowtie d$,

$$\forall \mathcal{X} \in \mathit{RS}(c), \mathcal{Y} \in \mathit{RS}(d).\ \big(\{\bar{\mathsf{a}} \mid \mathsf{a} \in \mathcal{X}\} \cap \mathcal{Y} \neq \emptyset \ \text{ or }\ \exists \mathsf{a}.\ \mathit{ready}\ \mathsf{a} \in (\mathcal{X} \cup \mathcal{Y}) \setminus (\mathcal{X} \cap \mathcal{Y})\big)$$

We now briefly review CO$_2$. Let $\mathcal{V}$ and $\mathcal{N}$ be disjoint sets of, respectively, *session variables* $x, y, \ldots$ and *session names* $s, t, \ldots$. Let $u, v, \ldots$ range over $\mathcal{V} \cup \mathcal{N}$. *Systems* S, *processes* P, *prefixes* π, and *latent contracts* K are defined below.

Definition 1 (CO$_2$ syntax). *The syntax of CO$_2$ is given by:*

$$\begin{array}{ll} P ::= \sum_i \pi_i.P_i \mid P|P \mid (\vec{u})P \mid X(\vec{u}) & \pi ::= \tau \mid \mathsf{tell}_{\mathsf{A}} {\downarrow_u} c \mid \mathsf{fuse} \mid \mathsf{do}_u \mathsf{a} \mid \mathsf{ask}_u \phi \\ K ::= {\downarrow_u}\ \mathsf{A}\, \mathit{says}\, c \mid K|K & S ::= \mathbf{0} \mid \mathsf{A}[P] \mid \mathsf{A}[K] \mid s[\gamma] \mid S|S \mid (\vec{u})S \end{array}$$

Processes specify the behaviour of participants. A process can be a prefix-guarded finite sum $\sum_i \pi_i.P_i$, a parallel composition $P \mid Q$, a delimited process $(\vec{u})P$, or a constant $X(\vec{u})$. We write $\mathbf{0}$ for $\sum_\emptyset P$, and $\pi_1.Q_1 + P$ for $\sum_{i \in I \cup \{1\}} \pi_i.Q_i$ provided that $P = \sum_{i \in I} \pi_i.Q_i$ and $1 \notin I$. We omit trailing occurrences of $\mathbf{0}$. We stipulate that each X has a unique defining equation $X(u_1, \ldots, u_j) \stackrel{\text{def}}{=} P$ s.t. $\text{fv}(P) \subseteq \{u_1, \ldots, u_j\} \subseteq \mathcal{V}$, and each constant occurring in P is prefix-guarded.

Prefixes include the silent action τ, contract advertisement $\mathsf{tell}_{\mathsf{A}} \downarrow_u c$, contract stipulation fuse, action execution $\mathsf{do}_u\ \mathsf{a}$, and contract query $\mathsf{ask}_u\ \phi$. In each prefix $\pi \neq \tau$, the identifier u refers to the target session involved in the execution of π. As in [3], we leave the syntax of *observables* ϕ unspecified.

A *latent contract* $\downarrow_x \mathsf{A}$ *says* c represents a contract c advertised by A but not stipulated yet. The variable x will be instantiated to a fresh session name upon stipulation. K simply stands for the parallel composition of latent contracts.

A system is composed of participants $\mathsf{A}[P]$, sessions $s[\gamma]$, sets of latent contracts advertised to A (denoted by $\mathsf{A}[K]$), and delimited systems $(\vec{u})S$. Delimitation $(\vec{u})$ binds session variables and names, both in processes and systems. Free variables and names are defined as usual, and denoted by $\text{fv}(\_)$ and $\text{fn}(\_)$. A system/process is *closed* when it has no free variables. Each participant may have at most one process, i.e. we forbid systems of the form $\mathsf{A}[P] \mid \mathsf{A}[Q]$. We say that S is A-*free* when it does not contain the participant $\mathsf{A}[P]$, nor latent contracts of A, nor sessions with A's contracts. Note that sessions cannot contain latent contracts.

The semantics of CO_2 is formalised by a reduction relation on systems (Def. 2). This relies on a standard structural congruence relation — of which we just point out that $(\vec{u})\mathsf{A}[(\vec{v})P] \equiv (\vec{u}\vec{v})\mathsf{A}[P]$ allows to move delimitations between systems and processes, while $\mathsf{A}[K] \mid \mathsf{A}[K'] \equiv \mathsf{A}[K \mid K']$ allows A to collect latent contracts.

In order to define honesty in § 3, here we decorate transitions with labels, by writing $\xrightarrow{\mathsf{A}\,:\,\pi,\sigma}$ for a reduction where participant A fires prefix π. Also, σ is a substitution which accounts for the instantiation of session variables upon a fuse.

Definition 2 ($\mathbf{CO_2}$ semantics). *The relation* $\xrightarrow{\mathsf{A}\,:\,\pi,\sigma}$ *between systems (considered up-to structural congruence* $\equiv$*) is the smallest relation closed under the rules of Fig. 1. The relation* $K \rhd^{\sigma} \gamma$ *holds iff* (*i*) K *has the form* $\downarrow_x \mathsf{A}$ *says* $c \mid \downarrow_y \mathsf{B}$ *says* d, (*ii*) $c \bowtie d$, (*iii*) $\gamma = \mathsf{A}$ *says* $c \mid \mathsf{B}$ *says* d, *and* (*iv*) $\sigma = \{s/_{x,y}\}$ *maps* $x, y \in \mathcal{V}$ *to* $s \in \mathcal{N}$. *The substitution* $\sigma_{\neq u}$ *in rule* [Del2] *is defined as* $\sigma(v)$ *for all* $v \neq u$, *and it is undefined on* u.

The rules in Fig. 1 are a minor variation of those presented in [3]. Their intuitive meaning is sketched in the introductory example (Sec. 1): [Tell] advertises a contract c, [Fuse] creates a new session s upon contractual compliance, [Do] performs a contractual action, [Ask] blocks until session s satisfies observable ϕ.

Example 1. A possible execution of $S = \mathsf{A}[(x)X(x)] \mid \mathsf{B}[(y)Y(y)] \mid \mathsf{C}[\mathsf{fuse}]$, where $X(x) \stackrel{\text{def}}{=} \mathsf{tell}_{\mathsf{C}} \downarrow_x (\mathsf{a}\ ;\ E) \,.\, \mathsf{do}_x\ \mathsf{a}$ and $Y(y) \stackrel{\text{def}}{=} \mathsf{tell}_{\mathsf{C}} \downarrow_y (\bar{\mathsf{a}} \,.\, E) \,.\, \mathsf{do}_y\ \bar{\mathsf{a}}$, is:

$$
\begin{aligned}
S &\xrightarrow{\mathsf{B}:\, \mathsf{tell}_{\mathsf{C}} \downarrow_y \bar{\mathsf{a}}, \emptyset} \mathsf{A}[(x)X(x)] \mid \mathsf{C}[\mathsf{fuse}] \mid (y)\,(\mathsf{B}[\mathsf{do}_y\ \bar{\mathsf{a}}] \mid \mathsf{C}[\downarrow_y \mathsf{B}\ \textit{says}\ \bar{\mathsf{a}} \,.\, E])\\
&\xrightarrow{\mathsf{A}:\, \mathsf{tell}_{\mathsf{C}} \downarrow_x \mathsf{a}, \emptyset} (x)(\mathsf{A}[\mathsf{do}_x\ \mathsf{a}] \mid (y)(\mathsf{B}[\mathsf{do}_y\ \bar{\mathsf{a}}] \mid \mathsf{C}[\mathsf{fuse}] \mid \mathsf{C}[\downarrow_x \mathsf{A}\ \textit{says}\ \mathsf{a}; E \mid \downarrow_y \mathsf{B}\ \textit{says}\ \bar{\mathsf{a}}.E]))\\
&\xrightarrow{\mathsf{C}:\, \mathsf{fuse}, \emptyset} (s)\,(\mathsf{A}[\mathsf{do}_s\ \mathsf{a}] \mid (y)\,(\mathsf{B}[\mathsf{do}_s\ \bar{\mathsf{a}}] \mid \mathsf{C}[\mathbf{0}] \mid s[\mathsf{A}\ \textit{says}\ \mathsf{a}; E \mid \mathsf{B}\ \textit{says}\ \bar{\mathsf{a}}.E]))\\
&\xrightarrow{\mathsf{A}:\, \mathsf{do}_s\ \mathsf{a}, \emptyset} (s)\,(\mathsf{A}[\mathbf{0}] \mid \mathsf{B}[\mathsf{do}_s\ \bar{\mathsf{a}}] \mid s[\mathsf{A}\ \textit{says}\ E \mid \mathsf{B}\ \textit{says}\ \textit{ready}\ \bar{\mathsf{a}} \,.\, E])
\end{aligned}
$$

$$\mathsf{A}[\tau.P + P' \mid Q] \xrightarrow{\mathsf{A}\,:\,\tau,\emptyset} \mathsf{A}[P \mid Q] \quad \text{[Tau]} \qquad \frac{S \xrightarrow{\mathsf{A}\,:\,\pi,\sigma} S' \quad \operatorname{ran}\sigma \cap \operatorname{fn}(S'') = \emptyset}{S \mid S'' \xrightarrow{\mathsf{A}\,:\,\pi,\sigma} S' \mid S''\sigma} \quad \text{[Par]}$$

$$\mathsf{A}[\mathsf{tell}_{\mathsf{B}} \downarrow_u c.P + P' \mid Q] \xrightarrow{\mathsf{A}\,:\,\mathsf{tell}_{\mathsf{B}} \downarrow_u c,\emptyset} \mathsf{A}[P \mid Q] \mid \mathsf{B}[\downarrow_u \mathsf{A}\ \mathit{says}\ c] \quad \text{[Tell]}$$

$$\frac{K \rhd^{\sigma} \gamma \qquad \operatorname{ran}\sigma = \{s\} \qquad s \text{ fresh}}{\mathsf{A}[\mathsf{fuse}.P + P' \mid Q] \mid \mathsf{A}[K] \xrightarrow{\mathsf{A}\,:\,\mathsf{fuse},\sigma} \mathsf{A}[P \mid Q]\sigma \mid s[\gamma]} \quad \text{[Fuse]}$$

$$\frac{\gamma \xrightarrow{\mathsf{A}\ \mathit{says}\ \mathsf{a}}\!\!\!\!\rightarrow \gamma'}{\mathsf{A}[\mathsf{do}_s\,\mathsf{a}.P + P' \mid Q] \mid s[\gamma] \xrightarrow{\mathsf{A}\,:\,\mathsf{do}_s\,\mathsf{a},\emptyset} \mathsf{A}[P \mid Q] \mid s[\gamma']} \quad \text{[Do]}$$

$$\frac{\gamma \vdash \phi}{\mathsf{A}[\mathsf{ask}_s\,\phi.P + P' \mid Q] \mid s[\gamma] \xrightarrow{\mathsf{A}\,:\,\mathsf{ask}_s\,\phi,\emptyset} \mathsf{A}[P \mid Q] \mid s[\gamma]} \quad \text{[Ask]} \qquad \frac{S \xrightarrow{\mathsf{A}\,:\,\pi,\{s/x\}} S'}{(x)S \xrightarrow{\mathsf{A}\,:\,\pi,\emptyset} (s)S'} \quad \text{[Del1]}$$

$$\frac{S \xrightarrow{\mathsf{A}\,:\,\pi,\sigma} S' \quad u \notin \operatorname{ran}\sigma \quad \sigma_{\neq u} \neq \emptyset}{(u)S \xrightarrow{\mathsf{A}\,:\,\pi,\sigma_{\neq u}} (u)S'} \quad \text{[Del2]} \qquad \frac{X(\vec{u}) \overset{\text{def}}{=} P \qquad \mathsf{A}[P\{^{\vec{v}}/_{\vec{u}}\} \mid Q] \mid S \xrightarrow{\mathsf{A}\,:\,\pi,\sigma} S'}{\mathsf{A}[X(\vec{v}) \mid Q] \mid S \xrightarrow{\mathsf{A}\,:\,\pi,\sigma} S'} \quad \text{[Def]}$$

Fig. 1. Reduction semantics of CO$_2$

3 On Honesty

A participant A is honest when she *realizes* every contract she advertises, in every session she may be engaged in. If a system S contains a session s with a contract c advertised by A, such as $\mathsf{A}[P] \mid s[\mathsf{A}\ \mathit{says}\ c \mid \cdots] \mid \cdots$, then A must realize c, even in a system populated by adversaries who play to cheat her. To realize c, A must be "ready" to behave according to c. For instance, if $\mathsf{A}[P]$ has advertised a contract c with an internal choice $c_i = \mathsf{a} \oplus \mathsf{b}$, then P must be ready to do *at least one* of the actions a, b. Instead, if c is an external choice $c_e = \mathsf{a} + \mathsf{b}$, then P must be ready to do *both* the actions a and b.

Realizability requires the *readiness* property to be preserved by all transitions of S. In other words, in any reduct of S containing a reduct P' of P and a reduct c' of c, the process P' must be ready for c'. To formalise when "P is ready for c" we inspect the ready sets $RS(c)$, which reveal whether c is exposing an internal or an external choice. At the process level, we consider the reachable actions in P.

Example 2. Let $c_i = \mathsf{a} \oplus \mathsf{b}$, and let $c_e = \mathsf{a} + \mathsf{b}$. Then, $RS(c_i) = \{\{\mathsf{a}\}, \{\mathsf{b}\}\}$, and $RS(c_e) = \{\{\mathsf{a}, \mathsf{b}\}\}$. Consider now the following processes:

– $P_0 = \mathsf{do}_s\,\mathsf{a}$ is ready for c_i, because $\{\mathsf{a}\} \in RS(c_i)$ and $\mathsf{do}_s\,\mathsf{a}$ is enabled in P_0. Instead, P_0 is *not* ready for c_e, since the ready set $\{\mathsf{a}, \mathsf{b}\}$ of c_e also contains b, which is not enabled in P_0.
– $P_1 = \mathsf{do}_s\,\mathsf{a} + \mathsf{do}_s\,\mathsf{b} + \mathsf{do}_s\,\mathsf{z}$ is ready for both c_i and c_e. Indeed, P_1 enables $\mathsf{do}_s\,\mathsf{a}$ and $\mathsf{do}_s\,\mathsf{b}$, thus covering the ready sets of c_i and c_e. The branch $\mathsf{do}_s\,\mathsf{z}$ is immaterial: rule [Do] blocks actions not expected by the contract.
– $P_2 = \tau.\mathsf{do}_s\,\mathsf{a} + \tau.\mathsf{do}_s\,\mathsf{b}$ is ready for c_i, because whatever branch is taken by P_2, it leads to an unguarded action which covers one of the ready sets in

c_i. Instead, P_2 is *not* ready for c_e, because after one of the two branches is chosen, one of the two actions expected by c_e is no longer available.

– $P_3 = \mathsf{do}_t\,\mathsf{w}.\mathsf{do}_s\,\mathsf{a} + \mathsf{do}_t\,\mathsf{z}.\mathsf{do}_s\,\mathsf{b}$ is a bit more complex than the above cases. Readiness w.r.t. c_i depends on the context. If the context eventually enables one of the do_t, then either $\mathsf{do}_s\,\mathsf{a}$ or $\mathsf{do}_s\,\mathsf{b}$ will be enabled, hence P_3 is ready for c_i. Otherwise, P_3 is stuck, hence it is not ready for c_i. Notice that P_3 is not ready for c_e, regardless of the context.

To formalise readiness, we start by defining the set $RD^{\mathsf{A}}_u(S)$ (for "Ready Do"), which collects all the atoms with an unguarded action do_u of A in a system S.

Definition 3 (Ready do). *For all S, A and u, we define the set $RD^{\mathsf{A}}_u(S)$ as:*

$$RD^{\mathsf{A}}_u(S) = \{\mathsf{a} \mid \exists \vec{v}, P, P', Q, S' \;.\; S \equiv (\vec{v})\,(\mathsf{A}[\mathsf{do}_u\,\mathsf{a}.P + P' \mid Q] \mid S') \;\wedge\; u \notin \vec{v}\}$$

As seen for P_2 and P_3, readiness may also hold when the actions expected in the contract ready sets are not immediately enabled in the process. To check if $\mathsf{A}[P]$ is ready for s (in a system S), we need to consider all actions which (1) are exposed in P after some steps, taken by P itself or by the context, and (2) are not preceded by other do_s performed by A. These actions form the set $WRD^{\mathsf{A}}_s(S)$.

Definition 4 (Weak ready do). *We define the set of atoms $WRD^{\mathsf{A}}_u(S)$ as:*

$$WRD^{\mathsf{A}}_u(S) \;=\; \{\mathsf{a} \mid \exists S' : S \xrightarrow{\neq(\mathsf{A}:\,\mathsf{do}_u)}{}^{*} S' \text{ and } \mathsf{a} \in RD^{\mathsf{A}}_u(S')\}$$

where $S \xrightarrow{\neq(\mathsf{A}:\,\mathsf{do}_u)} S'$ *iff* $\exists \mathsf{B}, \pi, \sigma.\;\; S \xrightarrow{\mathsf{B}:\,\pi,\sigma} S' \;\wedge\; (\mathsf{B} \neq \mathsf{A} \;\vee\; \forall \mathsf{a}.\, \pi \neq \mathsf{do}_u\,\mathsf{a})$.

Example 3. For $S \;=\; \mathsf{A}[\mathsf{do}_x\,\bar{\mathsf{a}}\,.\,\mathsf{do}_y\,\mathsf{b} + \tau\,.\,\mathsf{do}_y\,\mathsf{a}\,.\,\mathsf{do}_y\,\mathsf{c} \mid (x)\,\mathsf{do}_x\,\bar{\mathsf{b}}]$, we have:

$$WRD^{\mathsf{A}}_x(S) = \{\bar{\mathsf{a}}\} = RD^{\mathsf{A}}_x(S) \qquad\qquad WRD^{\mathsf{A}}_y(S) = \{\mathsf{a}\} \supseteq RD^{\mathsf{A}}_y(S) = \emptyset$$

On channel y, the action a is weakly reachable through its τ prefix. Action b is *not* weakly reachable, because guarded by a stuck do_x. Action c is *not* weakly reachable as well, because preceded by another do on the same channel.

Example 4. Let P_3 as in Ex. 2, and consider the following system:

$$\begin{aligned} S = \mathsf{A}[P_3] \;\mid\; \mathsf{B}[\tau\,.\,\mathsf{do}_s\,\bar{\mathsf{a}}\,.\,\mathsf{do}_s\,\bar{\mathsf{b}}] \;\mid\; \mathsf{C}[\mathsf{do}_t\,\overline{\mathsf{w}} + \mathsf{do}_t\,\bar{\mathsf{z}} + \tau] \\ \mid\; s[\mathsf{A}\;\mathit{says}\;\mathsf{a}+\mathsf{b} \mid \mathsf{B}\;\mathit{says}\;\bar{\mathsf{a}} \oplus \bar{\mathsf{b}}] \;\mid\; t[\mathsf{A}\;\mathit{says}\;\mathsf{w}+\mathsf{z} \mid \mathsf{C}\;\mathit{says}\;\overline{\mathsf{w}} \oplus \bar{\mathsf{z}}] \end{aligned}$$

In session t, A is immediately ready to do either w or z, so her ready do set coincides with her weak ready do set in t. The same for C, with the dual atoms $\overline{\mathsf{w}}$ and $\bar{\mathsf{z}}$. Thus:

$$WRD^{\mathsf{A}}_t(S) \;=\; RD^{\mathsf{A}}_t(S) \;=\; \{\mathsf{w},\mathsf{z}\} \qquad\qquad WRD^{\mathsf{C}}_t(S) \;=\; RD^{\mathsf{C}}_t(S) \;=\; \{\overline{\mathsf{w}},\bar{\mathsf{z}}\}$$

In session s, the ready do sets of both A and B are empty, because their actions are not immediately enabled. Before they can be reached, the whole system S must first reduce, either with the contribution of C on session t (in the case of A), or through a τ action (in the case of B). These reductions fall within the definition of their *weak* ready do sets, which are accordingly non-empty.

$$\mathit{WRD}_s^{\mathsf{B}}(S) = \{\bar{\mathsf{a}}\} \supseteq \mathit{RD}_s^{\mathsf{B}}(S) = \emptyset \qquad \mathit{WRD}_s^{\mathsf{A}}(S) = \{\mathsf{a}, \mathsf{b}\} \supseteq \mathit{RD}_s^{\mathsf{A}}(S) = \emptyset$$

Notice that $\bar{\mathsf{b}} \not\in \mathit{WRD}_s^{\mathsf{B}}(S)$: in fact, $\bar{\mathsf{b}}$ is guarded by $\mathsf{do}_s\ \bar{\mathsf{a}}$. Also, if C chooses to perform τ, then the actions in $\mathit{WRD}_s^{\mathsf{A}}(S)$ would not be reached. Indeed, Def. 4 requires that each element in the set becomes reachable at the end of a suitable reduction trace — but it does not prevent S from reducing along other paths.

A participant A is ready in a system S with a session $s[\mathsf{A}\ \mathit{says}\ c \mid \cdots]$ iff A is (weakly) ready to do all the actions in some ready set of c. Note that A is vacuously ready in systems not containing sessions with A's contracts.

Definition 5 (Honesty). *We say that* A *is ready in* S *iff, whenever* $S \equiv (\vec{u})S'$ *for some* $\vec{u}$ *and* $S' = s[\mathsf{A}\ \mathit{says}\ c \mid \cdots] \mid S_0$*,*

$$\exists \mathcal{X} \in \mathit{RS}(c)\,.\,\forall \mathsf{a} \neq \mathsf{e}\,.\,\big(\mathsf{a} \in \mathcal{X} \vee \mathit{ready}\,\mathsf{a} \in \mathcal{X} \implies \mathsf{a} \in \mathit{WRD}_s^{\mathsf{A}}(S')\big)$$

$\mathsf{A}[P]$ *is* honest *iff* $\forall$ A*-free* S *and* $\forall$ S' *such that* $\mathsf{A}[P] \mid S \rightarrow^* S'$*,* A *is ready in* S'*.*

The A-freeness requirement in Def. 5 is used just to rule out those systems already carrying stipulated or latent contracts of A outside $\mathsf{A}[P]$, e.g. $\mathsf{A}[P] \mid \mathsf{B}[\downarrow_x \mathsf{A}\ \mathit{says}\ \overline{\mathsf{pay}} \mid \cdots]$. In the absence of A-freeness, the system could trivially make $\mathsf{A}[P]$ dishonest.

Example 5. In the system below, A might look honest, although she is not.

$$S \stackrel{\text{def}}{=} \mathsf{A}[(x,y)\ (P_A \mid \mathsf{fuse} \mid \mathsf{fuse})] \mid \mathsf{B}[P_B] \mid \mathsf{C}[P_C]$$
$$P_A \stackrel{\text{def}}{=} \mathsf{tell}_{\mathsf{A}}\,(\downarrow_x \mathsf{a}\,.\,E)\,.\,\mathsf{tell}_{\mathsf{A}}\,(\downarrow_y \mathsf{b}\,;\,E)\,.\,\mathsf{do}_x\,\mathsf{a}\,.\,\mathsf{do}_y\,\mathsf{b}$$
$$P_B \stackrel{\text{def}}{=} (z)\ (\mathsf{tell}_{\mathsf{A}}\,(\downarrow_z \bar{\mathsf{b}}\,.\,E)\,.\,\mathsf{do}_z\,\bar{\mathsf{b}}) \qquad P_C \stackrel{\text{def}}{=} (w)\ (\mathsf{tell}_{\mathsf{A}}\,(\downarrow_w \bar{\mathsf{a}}\,;\,E)\,.\,\mathbf{0})$$

In fact, if we reduce S by performing all the tell and fuse actions, we obtain:

$$S' = (s,t)\,(\ \mathsf{A}[\mathsf{do}_t\,\mathsf{a}\,.\,\mathsf{do}_s\,\mathsf{b}] \mid \mathsf{B}[\mathsf{do}_s\,\bar{\mathsf{b}}] \mid \mathsf{C}[\mathbf{0}] \mid t[\mathsf{A}\ \mathit{says}\ \mathsf{a}\,.\,E \mid \mathsf{C}\ \mathit{says}\ \bar{\mathsf{a}}\,;\,E] \mid s[\mathsf{A}\ \mathit{says}\ \mathsf{b}\,;\,E \mid \mathsf{B}\ \mathit{says}\ \bar{\mathsf{b}}\,.\,E]\ )$$

Here, S' cannot reduce further: A is stuck, waiting for $\bar{\mathsf{a}}$ from C, which (dishonestly) avoids to do the required internal choice. So, A is dishonest, because she does not perform the promised b. Formally, A is dishonest because $\mathit{RS}(\mathsf{b}\,;\,E) = \{\{\mathsf{b}\}\}$, but $\mathsf{b} \not\in \mathit{WRD}_s^{\mathsf{A}}(S')$. Thus, A is not ready in S', hence not honest in S.

Our definition of honesty subsumes a *fair* scheduler, eventually allowing participants to fire persistently (weakly) enabled do actions. For instance, let $c = \mathsf{a} \oplus \mathsf{b}$, and let:

$$P \stackrel{\text{def}}{=} (x)\big(\mathsf{tell}_{\mathsf{A}} \downarrow_x c\,.\,\mathsf{fuse}\,.\,X(x)\big) \quad \text{where } X(x) \stackrel{\text{def}}{=} \tau\,.\,X(x) + \tau\,.\,\mathsf{do}_x\,\mathsf{a} + \tau\,.\,\mathsf{do}_x\,\mathsf{b}$$

Let $S = \mathsf{A}[P] \mid S_0$, and assume that the fuse in P passes. Under an unfair scheduler, A could always take the first branch in X, while neglecting the others. Intuitively, this would make A not respect her contract, which expects a or b. However, a fair scheduler will eventually choose one of the other branches.

4 A Type System for CO_2

We now introduce a type system for CO_2. The main result is *type safety* (established in Th. 4), which guarantees that typeable participants are honest.

The type of a process P is a function f, which maps each channel to a *channel type*. Channel types are behavioural types which essentially preserve the structure of P, while abstracting the actual prefixes and delimitations. Mainly, the prefixes of channel types distinguish between nonblocking and possibly blocking actions.

4.1 Channel Types

Channel types are Basic Parallel Processes (BPPs [14]) with standard semantics. Their prefixes can be atoms $(\mathsf{a}, \mathsf{b}, \ldots)$, contract advertisement actions $(\langle c \rangle)$, nonblocking (τ), possibly blocking $(\tau_?)$, and conditional (τ_ϕ) silent actions.

Definition 6 (Channel types). Channel types T *and* prefixes α *are:*

$$T \ ::= \ \mathbf{0} \mid \alpha \, . \, T \mid T + T \mid T \mid T \mid \mathit{rec}\ X \, . \, T \mid X \qquad \alpha \ ::= \ \mathsf{a} \mid \tau \mid \tau_? \mid \tau_\phi \mid \langle c \rangle$$

Example 6. Let $P = \mathsf{tell}_\mathsf{B} \downarrow_x c_i \mid (\mathsf{tell}_\mathsf{B} \downarrow_y d \, . \, \mathsf{do}_x \, \bar{\mathsf{a}})$, where $c_i = \bar{\mathsf{a}} \oplus \bar{\mathsf{b}}$, and d is immaterial. We anticipate that the channel types associated by our type system to P on channels x and y are, respectively, $T_x = \langle c_i \rangle \mid \tau . \bar{\mathsf{a}}$, and $T_y = \tau \mid \langle d \rangle . \tau_?$. Note that the advertisement of $\downarrow_x c_i$ is recorded in T_x, while that of $\downarrow_y d$ is abstracted there as a τ. Instead, the $\tau_?$ in T_y represents the fact that $\mathsf{do}_x \, \bar{\mathsf{a}}$ is not visible from channel y, and may potentially block.

The execution of CO_2 systems relies both on processes and contracts. Thus, we use an abstraction of the contract/process interplay to define abstract processes.

Definition 7 (Abstract processes). *An* abstract process *is a pair* (C, T) *or* (c, T)*, where* C *is a set of contracts,* c *is a contract, and* T *is a channel type.*

An abstract process (C, T) represents a process abstracted by T on some channel x, after the contracts in C have been *advertised*. Instead, (c, T) represents a process abstracted by T on channel x, after the contract c has been *stipulated*.

The semantics of abstract processes is given in Fig. 2. The set C grows when a channel type T in (C, T) performs a transition with label $\langle c \rangle$. After a contract $c \in C$ has been stipulated, the set is reduced to c. A label a models a $\mathsf{do}\,\mathsf{a}$ action performed by T, while rule *ctx* models an (unknown) action performed by the context. Further advertisements after contract stipulation are neglected.

The relation $\twoheadrightarrow_\sharp$ abstracts the contract semantics $\twoheadrightarrow$, by considering only the contract advertised by P (instead of the whole bilateral contract). We leave the relation $\twoheadrightarrow_\sharp$ unspecified (see [3] for a possible instantiation), and we just require that $\twoheadrightarrow_\sharp$ is decidable, and for all $\gamma = \mathsf{A}\ \mathit{says}\ c \mid \mathsf{B}\ \mathit{says}\ d$ such that $c \bowtie d$,

$$\gamma \xrightarrow{\mathsf{A}\ \mathit{says}\ \mathsf{a}} \mathsf{A}\ \mathit{says}\ c' \mid \mathsf{B}\ \mathit{says}\ d' \quad \Longrightarrow \quad c \xrightarrow{\mathsf{a}}_\sharp c'$$
$$\gamma \xrightarrow{\mathsf{B}\ \mathit{says}\ \mathsf{b}} \mathsf{A}\ \mathit{says}\ c' \mid \mathsf{B}\ \mathit{says}\ d' \quad \Longrightarrow \quad c \xrightarrow{\mathit{ctx}}_\sharp c'$$

$$\frac{T \xrightarrow{\langle c \rangle} T'}{(C,T) \to (C \cup \{c\}, T')} \qquad \frac{T \xrightarrow{\langle d \rangle} T'}{(c,T) \to (c,T')} \qquad \frac{c \in C}{(C,T) \to (c,T)} \qquad \frac{c \overset{ctx}{\twoheadrightarrow}_{\sharp} c'}{(c,T) \to (c',T)}$$

$$\frac{T \xrightarrow{\alpha} T' \quad \alpha \in \{\tau, \tau_?, \tau_\phi\}}{(C,T) \to (C,T')} \qquad \frac{c \overset{\mathsf{a}}{\twoheadrightarrow}_{\sharp} c' \quad T \xrightarrow{\mathsf{a}} T'}{(c,T) \to (c',T')} \qquad \frac{T \xrightarrow{\alpha} T' \quad \alpha \in \{\tau, \tau_?, \tau_\phi\}}{(c,T) \to (c,T')}$$

$$\frac{T \xrightarrow{\mathsf{a}} T'}{T \xRightarrow{\mathsf{a}} T'} \qquad \frac{T \xrightarrow{\tau} T'' \xRightarrow{\mathsf{a}} T'}{T \xRightarrow{\mathsf{a}} T'} \qquad \frac{T \xrightarrow{\langle d \rangle} T'' \xRightarrow{\mathsf{a}} T'}{T \xRightarrow{\mathsf{a}} T'} \qquad \frac{T \xrightarrow{\tau_\phi} T'' \xRightarrow{\mathsf{a}} T' \quad c \vdash^{\mathsf{A}}_{\sharp} \phi}{T \xRightarrow{\mathsf{a}} T'}$$

Fig. 2. Abstract processes semantics and channel type semantics

We now introduce the abstract counterpart of the dynamic notion of honesty in § 3. We shall follow the path outlined for concrete processes: first we define when a channel type T is "ready for a contract", and then when T is honest.

Weak transitions abstract the weak ready do. They are defined in Fig. 2 as a labelled relation $\xRightarrow{\mathsf{a}}{}^{\mathsf{A}}_{c}$ (simply written as $\xRightarrow{\mathsf{a}}$ when unambiguous). The first two rules are standard: they just collapse the τ actions as usual. The third rule also collapses contract advertisement actions, which are nonblocking as well. Possibly blocking actions $\tau_?$ are *not* collapsed, while τ_ϕ (which abstract $\mathsf{ask}_u\ \phi$ prefixes) are dealt with the last rule: they abstract the CO$_2$ prefix $\mathsf{ask}_u\ \phi$, and they are collapsed only if such ask is nonblocking. The relation $\vdash^{\mathsf{A}}_{\sharp}$ safely (under-) approximates this condition. We leave $\vdash^{\mathsf{A}}_{\sharp}$ unspecified (just like $\vdash$ in § 2), and we only require that it respects the constraint in Def. 8 below. Unlike in the concrete case, the context is immaterial in determining weak transitions.

Definition 8 (Abstract observability). *We write $c \vdash^{\mathsf{A}}_{\sharp} \phi$ for any decidable relation satisfying: $c \vdash^{\mathsf{A}}_{\sharp} \phi \implies \forall \mathsf{B} \,.\, \forall d \,.\, (c \bowtie d \implies \mathsf{A}$ says $c \mid \mathsf{B}$ says $d \vdash \phi)$.*

Below we relate the weak transitions of a channel type with the ready sets of a contract, similarly to what we did in Def. 5 for CO$_2$ processes. Weak transitions under-approximate the weak ready do set. Thus, if an abstract process is honest then also the concrete one will be such (while the *vice versa* is not always true).

Honesty of abstract processes is defined similarly to Def. 5. In order to be honest, a process must keep itself (abstractly) ready upon transitions. Readiness must be checked against all the contracts that may be stipulated along the abstract process reductions.

Definition 9 (Abstract honesty). *We say that channel type T is* abstractly ready for contract *c iff $\exists \mathcal{X} \in RS(c). \forall \mathsf{a} \neq \mathsf{e}. \big((\mathsf{a} \in \mathcal{X} \vee$ ready $\mathsf{a} \in \mathcal{X}) \implies T \xRightarrow{\mathsf{a}}\big)$. An abstract process $(-,T)$ is honest iff, whenever $(-,T) \to^* (c,T')$, then T' is ready for c. A channel type T is* honest *iff $(\emptyset, T)$ is honest.*

Example 7. Let $T_x = \langle c_i \rangle \mid \tau \,.\, \mathsf{a}$, and recall the contract $c_i = \mathsf{a} \oplus \mathsf{b}$ from Ex. 2. To prove T_x is honest, we examine all the reducts of the abstract process $(\emptyset, T_x)$ to check for readiness. We have the following cases:

1. $(\emptyset, T_x)$. Nothing to check, because no contracts have been advertised yet.
2. $(\emptyset, \langle c_i \rangle \mid \mathsf{a})$. Similar to the previous case.
3. $(\{c_i\}, \tau \,.\, \mathsf{a})$. Nothing to check, since no contracts have been stipulated yet.
4. $(\{c_i\}, \mathsf{a})$. Similar to the previous case.
5. $(c_i, \tau . \mathsf{a})$. Here $\tau . \mathsf{a}$ is ready for c_i, as $\{\mathsf{a}\} \in RS(c_i) = \{\{\mathsf{a}\}, \{\mathsf{b}\}\}$ and $\tau . \mathsf{a} \overset{\mathsf{a}}{\Longrightarrow}$.
6. (c_i, a). We have that a is ready for c_i, similarly to the previous case.
7. $(E, \mathbf{0})$. We have that $\mathbf{0}$ is vacuously ready for E.

Th. 1 below establishes that checking the honesty of a channel type T is decidable. Indeed, abstract readiness and abstract dishonesty are reachability properties. Abstract processes are the product of a finite state system (C and c only admit finitely many states), and a BPP T. This product can be modelled as a Petri net. Decidability follows since reachability is decidable for Petri nets [17].

Theorem 1. *Abstract honesty is decidable.*

4.2 Process Types

A *CO$_2$ process type* associates session names/variables to channel types, thus abstracting the behaviour of a process on all channels. Here, we introduce a "dummy" channel $* \notin \mathcal{N} \cup \mathcal{V}$, for collecting type information about unused channels. Formally, a process type is a function f from $\mathcal{N} \cup \mathcal{V} \cup \{*\}$ to channel types.

Our type system abstracts concrete prefixes of CO$_2$ processes as actions of channel types. We observe the behaviour of a process P on each channel, say u. When P performs an action on one of its channels, say v, we have two cases:

$v \neq u$: we will only observe a silent action, either nonblocking (τ) or blocking ($\tau_?$), depending on the concrete prefix fired.

$v = u$: we may observe more information, depending on the concrete prefix.

For instance, if P advertises a contract c with a $\mathsf{tell} \downarrow_v c$, then the action $\langle c \rangle$ will be visible if $v = u$, while we shall just observe a τ if $v \neq u$ (because tell is nonblocking). Similarly, if P performs $\mathsf{do}_v\, \mathsf{a}$ we shall observe the action a if $v = u$ and $\tau_?$ if $v \neq u$ (because do is blocking). Finally, if P executes a query $\mathsf{ask}_u\, \phi$ we shall observe the conditional silent action τ_ϕ if $v = u$ and $\tau_?$ otherwise.

Definition 10 (Prefix abstraction). *For all $u \in \mathcal{N} \cup \mathcal{V} \cup \{*\}$, we define the mapping $[\cdot]_u$ from CO$_2$ prefixes to channel type prefixes as follows:*

$$[\tau]_u = \tau \qquad [\mathsf{fuse}]_u = \tau_? \qquad [\mathsf{tell}_{\mathsf{A}} \downarrow_v c]_u = \texttt{if } v = u \texttt{ then } \langle c \rangle \texttt{ else } \tau$$

$$[\mathsf{do}_v\, \mathsf{a}]_u = \texttt{if } v = u \texttt{ then } \mathsf{a} \texttt{ else } \tau_? \qquad [\mathsf{ask}_v\, \phi]_u = \texttt{if } v = u \texttt{ then } \tau_\phi \texttt{ else } \tau_?$$

The typing judgments for processes have the form $\Gamma \vdash P : f$, where Γ is a typing environment, giving types to processes of the form $X(\vec{v})$.

Definition 11 (Typing environment). *A* typing environment Γ *is a partial function associating process types to constants $X(\vec{v})$.*

$$\frac{\Gamma \vdash P_i \colon f_i \quad \forall i \in I}{\Gamma \vdash \sum_{i\in I} \pi_i \,.\, P_i \colon \lambda u \,.\, \sum_{i\in I} [\pi_i]_u \,.\, f_i(u)}\ \text{[T-Sum]} \qquad \frac{\Gamma \vdash P \colon f \quad \Gamma \vdash Q \colon g}{\Gamma \vdash P \mid Q \colon \lambda u \,.\, f(u) \mid g(u)}\ \text{[T-Par]}$$

$$\frac{X(\vec{u}) \overset{\text{def}}{=} P \quad \Gamma\{f/X(\vec{v})\} \vdash P\{\vec{v}/\vec{u}\} \colon f}{\Gamma \vdash X(\vec{v}) \colon f}\ \text{[T-Def]} \qquad \frac{\Gamma(X(\vec{v})) = f}{\Gamma \vdash X(\vec{v}) \colon f}\ \text{[T-Var]}$$

$$\frac{\Gamma_{\neq u} \vdash P \colon f \quad f(u) \text{ honest}}{\Gamma \vdash (u)P \colon f\{f(*)/u\}}\ \text{[T-Del]} \quad \text{where } \Gamma_{\neq \vec{v}}(Y(\vec{w})) = \begin{cases} \Gamma(Y(\vec{w})) & \text{if } \vec{w} \cap \vec{v} = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

Fig. 3. Typing rules for processes

In Fig. 3 we introduce the typing rules for CO$_2$ processes. Rule [T-Sum] abstracts the prefixes which guard the branches of a summation, according to Def. 10. The resulting process type is expressed through the usual λ-notation. The type of a parallel composition is the pointwise parallel composition of the component types (rule [T-Par]). Rules [T-Def] and [T-Var] are mostly standard. Rule [T-Var] retrieves the type of a process variable from the typing environment, which is populated by [T-Def]. The rule for typing delimitations ([T-Del]) is worth some comments. Assume that P is typed with f. Since u in not free $(u)P$, the actions on channel u must not be observable in the typing of $(u)P$. To do that, in the typing of $(u)P$ we discard the information on u, by replacing it with the typing information on the "dummy" channel $*$. However, since this might hide a dishonest behaviour on u, the rule also checks that $f(u)$ is honest. Moreover, if the environment Γ has typing information on channel u, this cannot be used while typing P. The typing environment $\Gamma_{\neq u}$, which discards the information on u, is used to this purpose.

Example 8. Recall process $P_2 = \tau.\mathsf{do}_s\ \mathsf{a} + \tau.\mathsf{do}_s\ \mathsf{b}$ from Ex. 2. Its typing derivation is:

$$\frac{\dfrac{}{\vdash \mathsf{do}_s\ \mathsf{a} \colon \lambda u \,.\, [\mathsf{do}_s\ \mathsf{a}]_u = f_1}\ \text{[T-Sum]} \qquad \dfrac{}{\vdash \mathsf{do}_s\ \mathsf{b} \colon \lambda u \,.\, [\mathsf{do}_s\ \mathsf{b}]_u = f_2}\ \text{[T-Sum]}}{\vdash P_2 \colon f = \lambda u \,.\, [\tau]_u \,.\, f_1(u) + [\tau]_u \,.\, f_2(u)}\ \text{[T-Sum]}$$

We have $f(s) = \tau \,.\, \mathsf{a} + \tau \,.\, \mathsf{b}$, and for all $u \neq s$, $f(u) = f(*) = \tau \,.\, \tau_? + \tau \,.\, \tau_?$. In other words, the process type f performs some visible actions when observed at channel s, while remaining "silent" on other channels.

The type system assigns the same type (up-to structural congruence) to all non-free session names/variables, and to $*$; such type may only contain τ and $\tau_?$.

Lemma 1. *For all P, $\vdash P \colon f \implies f(*)$ only contains τ and $\tau_?$ actions.*

Lemma 2. $z \notin \text{fnv}(P) \ \wedge\ \Gamma \vdash P \colon f \implies f(z) = f(*)$

A process type f takes a transition on a CO$_2$ prefix π when all its points $f(u)$ agree to take a transition on the abstract prefix $[\pi]_u$.

Definition 12. *We write $f \xrightarrow{\pi} f'$ whenever $\forall u \in \mathcal{N} \cup \mathcal{V} \cup \{*\} \,.\, f(u) \xrightarrow{[\pi]_u} f'(u)$.*

Example 9. Recall $P_1 = \mathsf{do}_s\,\mathsf{a} + \mathsf{do}_s\,\mathsf{b} + \mathsf{do}_s\,\mathsf{z}$ from Ex. 2. Its typing is $\vdash P_1 \colon f = \lambda u\,.\,[\mathsf{do}_s\,\mathsf{a}]_u + [\mathsf{do}_s\,\mathsf{b}]_u + [\mathsf{do}_s\,\mathsf{z}]_u$. Let $f' = \lambda u\,.\,\mathbf{0}$. We have that $f \xrightarrow{\mathsf{do}_s\,\mathsf{a}} f'$, since:

- $[\mathsf{do}_s\,\mathsf{a}]_s = \mathsf{a}$ and $f(s) = \mathsf{a} + \mathsf{b} + \mathsf{z} \xrightarrow{\mathsf{a}} \mathbf{0} = f'(s)$;
- $\forall v \neq s\,.\,[\mathsf{do}_s\,\mathsf{a}]_v = \tau_?$ and $f(v) = \tau_? + \tau_? + \tau_? \xrightarrow{\tau_?} \mathbf{0} = f'(v)$.

Note that, in this case, we also have $f \xrightarrow{\mathsf{do}_s\,\mathsf{b}} f'$ and $f \xrightarrow{\mathsf{do}_s\,\mathsf{z}} f'$.

If f is the type of some CO$_2$ process (i.e., f is *inhabited*), and a single point $f(u)$ takes an abstract transition, then the whole f can take a transition.

Lemma 3. $f(u) \xrightarrow{\alpha} T' \wedge f\,\textit{inhabited} \implies \exists \pi, f'\,.\,[\pi]_u = \alpha \wedge f'(u) = T' \wedge f \xrightarrow{\pi} f'$.

Definition 13 (Type honesty). *f is* honest *iff $f(u)$ is honest, $\forall u \in \mathcal{N} \cup \mathcal{V} \cup \{*\}$.*

Note that, when $\vdash P \colon f$, checking the honesty of f amounts to checking $f(u)$ honest *only* for each $u \in \mathrm{fnv}(P)$. In fact, $f(v) = f(*)$ when $v \notin \mathrm{fnv}(P)$, and $f(*)$ is trivially honest because it cannot advertise contracts.

4.3 System Typing

In this section we establish *type safety* for CO$_2$ processes: for any typeable closed P, $\mathsf{A}[P]$ is honest. To prove this result, we have to consider the transitions of a process within a system. Hence, in order to construct an invariant of the system transitions (i.e., proving subject reduction), we extend typing to systems.

Type judgments for systems are of two kinds. A judgment of the form $\vdash_{\mathsf{A}} S \colon f$ guarantees that a participant A in S behaves according to f. Instead, a judgment of the form $\vdash_{\mathsf{A}} S \rhd f$ means that A is *not* in S, and S is *compatible* with any participant A behaving as f. Our notion of compatibility is liberal: intuitively, it just checks that the context S does not contain forged contracts of A.

Definition 14 (System typing). *The relations $\vdash_{\mathsf{A}} S \colon f$ and $\vdash_{\mathsf{A}} S \rhd f$ are the smallest relations closed under the rules in Fig. 4.*

Most rules in Fig. 4 are straightforward. Rules [T-SAFree*] tell that A-free systems are compatible with all f. Rules [T-SFz*] state that an f-compatible context (where f is the behaviour of A) may contain latent contracts of A if f realizes them. Rule [T-SFused] is similar, but it deals with stipulated contracts of A. Rule [T-SDel2] is similar to [T-Del] for typing processes. Rule [T-SDel1] is dual: while in [T-SDel2] the type f abstracts the behaviour of A *within* S, in [T-SDel1] it represents the behaviour of A *outside* S. Note that if $\mathsf{A}[P]$ is typeable, then it can be inserted in any A-free system, and the composed system will be typeable.

Example 10. Let $S_0 = \mathsf{B}[Q] \mid \mathsf{C}[\downarrow_x \mathsf{A}\ \textit{says}\ c]$, with $\mathsf{B} \neq \mathsf{A}$ (note that S_0 is not A-free). Assume that $\vdash P \colon f$. Then, a typing derivation of $S = \mathsf{A}[P] \mid S_0$ is:

$$\dfrac{\dfrac{\vdash P \colon f}{\vdash_{\mathsf{A}} \mathsf{A}[P] \colon f}\,\text{[T-SA]} \qquad \dfrac{\dfrac{\mathsf{B} \neq \mathsf{A}}{\vdash_{\mathsf{A}} \mathsf{B}[Q] \rhd f}\,\text{[T-SAFree1]} \qquad \dfrac{f(x)\ \text{realizes}\ c}{\vdash_{\mathsf{A}} \mathsf{C}[\downarrow_x \mathsf{A}\ \textit{says}\ c] \rhd f}\,\text{[T-SFz1]}}{\vdash_{\mathsf{A}} \mathsf{B}[Q] \mid \mathsf{C}[\downarrow_x \mathsf{A}\ \textit{says}\ c] = S_0 \rhd f}\,\text{[T-SPar1]}}{\vdash_{\mathsf{A}} \mathsf{A}[P] \mid S_0 = S \colon f}\,\text{[T-SPar2]}$$

Notice that S is typeable with f only if $f(x)$ realizes A's contract c.

$$\frac{}{\vdash_{\mathsf{A}} \mathbf{0} \rhd f}\,\text{[T-SAFree0]} \qquad \frac{\mathsf{B} \neq \mathsf{A}}{\vdash_{\mathsf{A}} \mathsf{B}[P] \rhd f}\,\text{[T-SAFree1]} \qquad \frac{\mathsf{B} \neq \mathsf{A}}{\vdash_{\mathsf{A}} \mathsf{C}[\downarrow_x \mathsf{B}\ \mathit{says}\ c] \rhd f}\,\text{[T-SAFree2]} \qquad \frac{\gamma\ \ \mathsf{A}\text{-free}}{\vdash_{\mathsf{A}} s[\gamma] \rhd f}\,\text{[T-SAFree3]}$$

$$\frac{}{\vdash_{\mathsf{A}} \mathsf{B}[\downarrow_s \mathsf{A}\ \mathit{says}\ c] \rhd f}\,\text{[T-SFzS]} \qquad \frac{f(x)\ \text{realizes}\ c}{\vdash_{\mathsf{A}} \mathsf{B}[\downarrow_x \mathsf{A}\ \mathit{says}\ c] \rhd f}\,\text{[T-SFz1]} \qquad \frac{\vdash_{\mathsf{A}} \mathsf{B}[K] \rhd f \quad \vdash_{\mathsf{A}} \mathsf{B}[K'] \rhd f}{\vdash_{\mathsf{A}} \mathsf{B}[K \mid K'] \rhd f}\,\text{[T-SFz2]}$$

$$\frac{\emptyset \vdash P\colon f}{\vdash_{\mathsf{A}} \mathsf{A}[P]\colon f}\,\text{[T-SA]} \qquad \frac{\vdash_{\mathsf{A}} S \rhd f\{^{f(*)}/_{u}\}}{\vdash_{\mathsf{A}} (u)S \rhd f}\,\text{[T-SDel1]} \qquad \frac{\vdash_{\mathsf{A}} S\colon f \quad f(u)\ \text{honest}}{\vdash_{\mathsf{A}} (u)S\colon f\{^{f(*)}/_{u}\}}\,\text{[T-SDel2]}$$

$$\frac{f(s)\ \text{realizes}\ c}{\vdash_{\mathsf{A}} s[\mathsf{A}\ \mathit{says}\ c \mid \cdots] \rhd f}\,\text{[T-SFused]} \qquad \frac{\vdash_{\mathsf{A}} S \rhd f \quad \vdash_{\mathsf{A}} S' \rhd f}{\vdash_{\mathsf{A}} S \mid S' \rhd f}\,\text{[T-SPar1]} \qquad \frac{\vdash_{\mathsf{A}} S\colon f \quad \vdash_{\mathsf{A}} S' \rhd f}{\vdash_{\mathsf{A}} S \mid S'\colon f}\,\text{[T-SPar2]}$$

Fig. 4. Typing rules for systems. Symmetric rules wrt | for [T-SFused], [T-SPar2] omitted.

4.4 Type Safety

Subject reduction guarantees that typeability is preserved by transitions. We need to distinguish two cases, according to which participant moves: either the participant A under typing, or any other B. If the transition is done by A, then also its process type must take a transition, otherwise the type is preserved.

The substitution induced when executing a fuse also affects the type of the reduct system. For instance, consider $\mathsf{A}[P] \mid S$, where $\vdash P\colon f$ and $f(x) = T$. Assume that S fires a fuse, with a substitution σ mapping x to a fresh session name s. In the type, this is reflected by updating f according to σ, so to map s to T, and x, which is no longer free, to $f(*)$. Technically, this is done as follows.

Definition 15. *For a type f and a mapping $\{^{v}/_{\vec{u}}\}$, the partial operator $\bullet$ is:*

$$f \bullet \{^{v}/_{\vec{u}}\} = \begin{cases} f & \text{if}\ \forall u_0 \in \vec{u}\,.\, f(u_0) = f(*) \\ f\{^{f(*)}/_{u_0}\}\{^{f(u_0)}/_{v}\} & \text{if}\ \exists! u_0 \in \vec{u}\,.\, f(u_0) \neq f(*) \end{cases}$$

Theorem 2 (Subject reduction). *If $\vdash_{\mathsf{A}} S\colon f$ with f honest, then:*

$$S \xrightarrow{\mathsf{A}\,:\,\pi,\sigma} S' \implies \exists f' \,.\, f \xrightarrow{\pi} f' \wedge \vdash_{\mathsf{A}} S'\colon f' \bullet \sigma$$
$$S \xrightarrow{\mathsf{B}\,:\,\pi,\sigma} S' \implies \vdash_{\mathsf{A}} S'\colon f \bullet \sigma \qquad (\textit{when}\ \mathsf{B} \neq \mathsf{A})$$

Progress guarantees that if a typeable process has a "non-blocking" type, then it can take a transition. More precisely, if the type of P on channel u can take a weak transition with label a, then P will have a in its weak ready do set.

Theorem 3 (Progress). *For all $S \equiv s[\mathsf{A}\ \mathit{says}\ c \mid \cdots] \mid S'$, if $\vdash_{\mathsf{A}} S\colon f$ with f honest, and $f(s) \overset{\mathsf{a}}{\Longrightarrow}{}^{\mathsf{A}}_{c}$, then $\mathsf{a} \in \mathit{WRD}^{\mathsf{A}}_{s}(S)$.*

The main result of this paper is the type safety of CO_2 processes. It ensures that a participant A with a well-typed process P will always respect her contracts — both those already advertised, and those that she will publish along her reductions. Therefore, A will never be considered culpable in any context.

Theorem 4 (Type safety). $\forall\ \mathsf{A}[P]$ *with P closed,* $\vdash P\colon f \implies \mathsf{A}[P]$ *is honest.*

5 Concluding Remarks and Related Work

We have given a type system for a calculus of contracting processes. Type safety establishes *honesty*: typeable processes honour their contracts in all contexts.

In [3], A is considered "honest" when not definitely *culpable* (in any session), i.e., A eventually performs the actions her contract prescribes. In Def. 5, honesty is based on readiness, rather than culpability; we conjecture that the two notions are equivalent. The main advantage of this new approach compared to [3] is that it simplifies the proof of the correctness of the static analysis of honesty, by more directly relating abstract transitions with concrete ones. Also, the new definition helps in proving decidability of abstract honesty, which was left open in [3].

In [5] (multiparty) asserted global types are used to adapt design-by-contract to distributed interactions. In our framework, a participant declares its contract independently of the others; a CO_2 primitive (fuse) tries then to combine advertised contracts within a suitable agreement. In other words, one could think of our approach as based on orchestration rather than choreography.

In [16] the progress property is checked only when participants engage at most in one session at a time. Honesty is a sort of progress property, and our type system allows participants to interleave many sessions as done in [15]. A crucial difference with respect to [15] is that the typing discipline there requires the *consistency* of the local types of any two participants interacting in a session. Namely, if in a session s, A and B are typed as T_{A} and T_{B} respectively and they interact then the projection of T_{A} with respect to B must be dual of the projection of T_{B} with respect to A. In our type system instead, participants are typechecked 'in isolation' and to establish the honesty of a participant A our typing discipline only imposes that the surrounding context is A-free.

Other approaches deal with safety, by generating monitors that check at runtime the interactions of processes against their local contract (e.g.,[13,12]).

The problem of checking if a contract c, representing the behaviour of a service, conforms to a role r of a choreography H has been investigated in [6]. Conformance of c is attained by establishing a *should testing* pre-order between c and the projection of H with respect to role r. Similar techniques have been used in [7] to define contract-based service composition. A main difference w.r.t. our approach, is that [6,7] do not consider conformance in the presence of dishonest participants. Actually, they focus on matching a contract as a role within a choreography, while we establish if a process abides by its own contracts, regardless of the context.

In [8,9], constraints are used to rule out "inconsistent" executions. This is orthogonal to our approach, since our aim is to blame participants that misbehave.

Acknowledgments. Work partially supported by Aut. Region of Sardinia under grants L.R.7/2007 CRP-17285 (TRICS), P.I.A. 2010 Project "Social Glue", and by MIUR PRIN 2010-11 project "Security Horizons", and by the Leverhulme Trust Programme Award "Tracing Networks", and by COST Action IC1201: Behavioural Types for Reliable Large-Scale Software Systems (BETTY), and by European Union Seventh Framework Programme under grant agreement no. 295261 (MEALS).

References

1. Bartoletti, M., Scalas, A., Tuosto, E., Zunino, R.: Honesty by typing. Technical report (2013), http://tcs.unica.it/publications
2. Bartoletti, M., Tuosto, E., Zunino, R.: Contract-oriented computing in CO_2. Scientific Annals in Computer Science 22(1), 5–60 (2012)
3. Bartoletti, M., Tuosto, E., Zunino, R.: On the realizability of contracts in dishonest systems. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 245–260. Springer, Heidelberg (2012)
4. Bartoletti, M., Zunino, R.: A calculus of contracting processes. In: LICS (2010)
5. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 162–176. Springer, Heidelberg (2010)
6. Bravetti, M., Zavattaro, G.: Towards a unifying theory for choreography conformance and contract compliance. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 34–50. Springer, Heidelberg (2007)
7. Bravetti, M., Zavattaro, G.: Contract-based discovery and composition of web services. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 261–295. Springer, Heidelberg (2009)
8. Buscemi, M.G., Coppo, M., Dezani-Ciancaglini, M., Montanari, U.: Constraints for service contracts. In: Bruni, R., Sassone, V. (eds.) TGC 2011. LNCS, vol. 7173, pp. 104–120. Springer, Heidelberg (2012)
9. Buscemi, M.G., Montanari, U.: CC-pi: A constraint-based language for specifying service level agreements. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 18–32. Springer, Heidelberg (2007)
10. 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)
11. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. ACM TOPLAS 31(5) (2009)
12. Chen, T.-C., Bocchi, L., Deniélou, P.-M., Honda, K., Yoshida, N.: Asynchronous distributed monitoring for multiparty session enforcement. In: Bruni, R., Sassone, V. (eds.) TGC 2011. LNCS, vol. 7173, pp. 25–45. Springer, Heidelberg (2012)
13. Chen, T.-C., Honda, K.: Specifying Stateful Asynchronous Properties for Distributed Programs. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 209–224. Springer, Heidelberg (2012)
14. Christensen, S.: Decidability and Decomposition in Process Algebras. PhD thesis, Edinburgh University (1993)
15. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global Progress for Dynamically Interleaved Multiparty Sessions (2013), http://www.di.unito.it/~padovani/Papers/CoppoDezaniYoshidaPadovani13.pdf
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL (2008)
17. Mayr, R.: Decidability and Complexity of Model Checking Problems for Infinite-State Systems. PhD thesis, Technische Universität München (1998)

Author Index