

Christel Baier
Luís Caires (Eds.)

LNCS 10854

Formal Techniques for Distributed Objects, Components, and Systems

38th IFIP WG 6.1 International Conference, FORTE 2018
Held as Part of the 13th International Federated Conference
on Distributed Computing Techniques, DisCoTec 2018
Madrid, Spain, June 18–21, 2018, Proceedings



ifip

 Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, Lancaster, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Friedemann Mattern

ETH Zurich, Zurich, Switzerland

John C. Mitchell

Stanford University, Stanford, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

C. Pandu Rangan

Indian Institute of Technology Madras, Chennai, India

Bernhard Steffen

TU Dortmund University, Dortmund, Germany

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbrücken, Germany

More information about this series at <http://www.springer.com/series/7408>


Christel Baier · Luís Caires (Eds.)

Formal Techniques for Distributed Objects, Components, and Systems

38th IFIP WG 6.1 International Conference, FORTE 2018
Held as Part of the 13th International Federated Conference
on Distributed Computing Techniques, DisCoTec 2018
Madrid, Spain, June 18–21, 2018
Proceedings

Editors

Christel Baier
Technische Universität Dresden
Dresden
Germany

Luís Caires 
Departamento de Informática
Universidade Nova de Lisboa
Caparica
Portugal

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-92611-7 ISBN 978-3-319-92612-4 (eBook)
<https://doi.org/10.1007/978-3-319-92612-4>

Library of Congress Control Number: 2018944400

LNCS Sublibrary: SL2 – Programming and Software Engineering

© IFIP International Federation for Information Processing 2018, corrected publication 2018

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by the registered company Springer International Publishing AG
part of Springer Nature
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Foreword

The 13th International Federated Conference on Distributed Computing Techniques (DisCoTec) took place in Madrid, Spain, during June 18–21, 2018. The DisCoTec series is one of the major events sponsored by the International Federation for Information Processing (IFIP). It comprises three conferences:

- COORDINATION, the IFIP WG6.1 International Conference on Coordination Models and Languages (the conference celebrated its 20th anniversary in 2018)
- DAIS, the IFIP WG6.1 International Conference on Distributed Applications and Interoperable Systems (the conference is in its 18th edition)
- FORTE, the IFIP WG6.1 International Conference on Formal Techniques for Distributed Objects, Components and Systems (the conference is in its 38th edition)

Together, these conferences cover a broad spectrum of distributed computing subjects, ranging from theoretical foundations and formal description techniques to systems research issues. Each day of the federated event began with a plenary speaker nominated by one of the conferences.

In addition to the three main conferences, two satellite events took place during June 20–21, 2018:

- ICE, the Workshop on Interaction and Concurrency Experience (in its 11th edition)
- FADL, Workshop on Foundations and Applications of Distributed Ledgers (this was the first year that the workshop took place)

I would like to thank the Program Committee chairs of the different events for their help and cooperation during the preparation of the conference and the Steering Committee of DisCoTec for its guidance and support. The organization of DisCoTec 2018 was only possible thanks to the dedicated work of the Organizing Committee, including the organization chairs, Jesús Correas and Sonia Estévez (Universidad Complutense de Madrid, Spain), the publicity chair, Ivan Lanese (University of Bologna/Inria, Italy), the workshop chairs, Luis Llana and Ngoc-Thanh Nguyen (Universidad Complutense de Madrid, Spain and Wroclaw University of Science and Technology, Poland, respectively), the finance chair, Mercedes G. Merayo (Universidad Complutense de Madrid, Spain), and the webmaster, Pablo C. Cañizares (Universidad Complutense de Madrid, Spain). Finally, I would like to thank IFIP WG6.1 for sponsoring this event, Springer’s *Lecture Notes in Computer Science* team for their support and sponsorship, and EasyChair for providing the reviewing infrastructure.

June 2018

Manuel Núñez

Preface

This volume contains the papers presented at the 38th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE 2018). It was held as one of the three main conferences of the 13th International Federated Conference on Distributed Computing Techniques (DisCoTec), June 18–21, 2018, in Madrid.

The conference is dedicated to fundamental research on theory, models, tools, and applications for distributed systems. It solicits original contributions that advance the science and technologies for distributed systems, with special interest in the areas of: component- and model-based design; object technology, modularity, software adaptation, service-oriented, ubiquitous, pervasive, grid, cloud, and mobile computing systems; software quality, reliability, availability, and safety; security, privacy, and trust in distributed systems; adaptive distributed systems; self-stabilization; self-healing/organizing; verification, validation, formal analysis, and testing of the above.

The program consisted of ten contributed papers, selected from 28 submissions. Each submission was reviewed by at least three Program Committee members, with the help of external experts. The selection was made based on discussions via the EasyChair conference management system, which was also used to assist with the assembly of the proceedings.

We wish to thank all authors who submitted to FORTE 2018, all the Program Committee members for their excellent work, and the external reviewers for their thorough evaluation of the submissions. We want to say a special thanks to Joachim Klein, who helped us to generate the conference proceedings. In addition, we would like to thank the DisCoTec Organizing Committee for providing an excellent environment for FORTE and other conferences and workshops.

April 2018

Christel Baier
Luís Caires

Organization

Program Committee

Christel Baier	Technische Universität Dresden, Germany
Luis Barbosa	University of Minho, Portugal
Dirk Beyer	Ludwig-Maximilian-Universität München (LMU), Germany
Nikolaj Bjorner	Microsoft Research, USA
Ahmed Bouajjani	IRIF, University of Paris Diderot, France
Tomas Brazdil	Masaryk University, Czech Republic
Luís Caires	Universidade NOVA de Lisboa, Portugal
Ilaria Castellani	Inria, France
Yuxin Deng	East China Normal University, China
Patrick Eugster	University of Lugano (USI), Switzerland
Arnd Hartmanns	University of Twente, The Netherlands
Thomas Hildebrandt	University of Copenhagen, Denmark
Nils Jansen	Radboud University Nijmegen, The Netherlands
Einar Broch Johnsen	University of Oslo, Norway
Antónia Lopes	Universidade de Lisboa, Portugal
Matteo Mio	CNRS/ENS-Lyon, France
Anna Philippou	University of Cyprus, Cyprus
Jorge A. Pérez	University of Groningen, The Netherlands
Davide Sangiorgi	University of Bologna, Italy
Mahsa Shirmohammadi	CNRS and LIS, France
Alexandra Silva	University College London, UK
Mahesh Viswanathan	University of Illinois at Urbana-Champaign, USA
Heike Wehrheim	Universität Paderborn, Germany
Verena Wolf	Universität des Saarlandes, Germany

Additional Reviewers

Bauer, Matthew	Din, Crystal Chang	Junges, Sebastian
Brunet, Paul	Dokter, Kasper	Kappe, Tobias
Capecchi, Sara	Friedberger, Karlheinz	Koutny, Maciej
Chen, Zhe	Gaspari, Mauro	Kouzapas, Dimitrios
Cheval, Vincent	Heam, Pierre-Cyrille	König, Jürgen
Dezani-Ciancaglini,	Jakobs, Marie-Christine	Lanese, Ivan
Mariangiola	Janssen, Ramon	Long, Huan

Madeira, Alexandre
Mathur, Umang
Peraldi-Frati,
Marie-Agnès
Proenca, Jose

Pun, Ka I
Sammartino, Matteo
Schlatte, Rudolf
Stan, Daniel
Toews, Manuel

van Dijk, Tom
Viering, Malte
Wimmer, Ralf
Worrell, James

Contents

A Distributed Coordination Infrastructure for Attribute-Based Interaction	1
<i>Yehia Abd Alrahman, Rocco De Nicola, Giulio Garbi, and Michele Loreti</i>	
Applied Choreographies	21
<i>Saverio Giallorenzo, Fabrizio Montesi, and Maurizio Gabbrielli</i>	
Monotonic Prefix Consistency in Distributed Systems	41
<i>Alain Girault, Gregor Gössler, Rachid Guerraoui, Jad Hamza, and Dragos-Adrian Seredinschi</i>	
A Modest Security Analysis of Cyber-Physical Systems: A Case Study	58
<i>Ruggero Lanotte, Massimo Merro, and Andrei Munteanu</i>	
Relating Process Languages for Security and Communication Correctness (Extended Abstract)	79
<i>Daniele Nantes and Jorge A. Pérez</i>	
A Calculus for Modeling Floating Authorizations	101
<i>Jovanka Pantović, Ivan Prokić, and Hugo Torres Vieira</i>	
Parameter Synthesis Algorithms for Parametric Interval Markov Chains	121
<i>Laure Petrucci and Jaco van de Pol</i>	
Information Flow Tracking for Side-Effectful Libraries	141
<i>Alexander Sjösten, Daniel Hedin, and Andrei Sabelfeld</i>	
On a Verification Framework for Certifying Distributed Algorithms: Distributed Checking and Consistency	161
<i>Kim Völlinger and Samira Akili</i>	
Preserving Contract Satisfiability Under Non-monotonic Composition	181
<i>Jonas Westman and Mattias Nyberg</i>	
Correction to: Applied Choreographies	E1
<i>Saverio Giallorenzo, Fabrizio Montesi, and Maurizio Gabbrielli</i>	
Author Index	197



A Distributed Coordination Infrastructure for Attribute-Based Interaction

Yehia Abd Alrahman¹(✉) , Rocco De Nicola¹ , Giulio Garbi¹ ,
and Michele Loreti² 

¹ IMT School for Advanced Studies Lucca, Lucca, Italy
yehia.abdalrahman@imtlucca.it

² Università di Camerino, Camerino, Italy

Abstract. Collective-adaptive systems offer an interesting notion of interaction where run-time contextual data are the driving force for interaction. The attribute-based interaction has been proposed as a foundational theoretical framework to model CAS interactions. The framework permits a group of partners to interact by considering their run-time properties and their environment. In this paper, we lay the basis for an efficient, correct, and distributed implementation of the attribute-based interaction framework. First, we present three coordination infrastructures for message exchange, then we prove their correctness, and finally we model them in terms of stochastic processes to evaluate their performance.

Keywords: Attribute-based interaction · Semantics · Process calculi

1 Introduction

Collective Adaptive Systems (CAS) [12] consists of a large number of components that interact anonymously, based on their properties and on contextual data, and combine their behaviours to achieve system-level goals. The boundaries of CAS are fluid and components may enter or leave the system at any time. Components may also adapt their behaviours in response to environmental conditions.

Classical communication paradigms handle the interaction among distributed components by relying on their identities, like in the Actor model [5], or on channel names, like in channel-based binary communication [18] and broadcast communication [16]. However, since identities and channels are totally independent from run-time properties and capabilities of the interacting components, programming collective-adaptive behaviour becomes a tedious task.

To mitigate the shortcomings of the classical paradigms when dealing with CAS, in FORTE'16 [6], we have proposed a kernel calculus, named *AbC* [1],

This research has been supported by the European projects IP 257414 ASCENS and STReP 600708 QUANTICOL.

for modeling CAS interactions. The idea is to permit the construction of formally verifiable CAS systems by relying on a minimal set of interaction primitives. *AbC*'s primitives are attribute-based [4] and abstract from the underlying coordination infrastructure (i.e., they are infrastructure-agnostic). They rely on *anonymous* multicast communication where components interact based on mutual interests. Message transmission is non-blocking while reception is not. Each component has a set of attributes to represent its run-time status. Communication actions (both send and receive) are decorated with predicates over attributes that partners have to satisfy to make the interaction possible. The interaction predicates are also parametrised with local attribute values and when values change, the interaction groups do implicitly change, introducing opportunistic interactions.

Basing the interaction on run-time attribute values is indeed a nice idea, but it needs to be supported by a middleware that provides efficient ways for distributing messages, checking attribute values, and updating them. A typical approach is to rely on a centralised broker that keeps track of all components, intercepts every message and forwards it to registered components. It is then the responsibility of each component to decide whether to receive or discard the message. This is the approach used in the Java-based implementation [2] of *AbC*. A similar approach, still based on a centralised broker, is used in the Erlang-based implementation [10]. There however to avoid broadcasts the broker has an attribute registry where components register their attribute values and the broker is now responsible for message filtering.

Clearly, any centralised solution may not scale with CAS dynamics and thus becomes a bottleneck for performance. A distributed approach is definitely preferable for large systems. However, distributed coordination infrastructures for managing the interaction of computational systems are still scarce [15] and/or inefficient [20]. Also the correctness of their overall behaviour is often not obvious. In this paper, we propose an efficient distributed coordination infrastructure for message exchange. We prove its correctness with respect to the original semantics of *AbC* and finally we evaluate its performance in terms of stochastic simulation. Though this paper assumes perfect communication links and does not deal with dropped messages or node's failures, we believe that existing techniques for resilience and failure-recovery can be integrated transparently.

The rest of this paper is structured as follows: In Sect. 2, we briefly review the *AbC* calculus. In Sect. 3, we give a full formal account of a distributed coordination infrastructure for *AbC* and its correctness. In Sect. 4, we provide a detailed performance evaluation and we discuss the results. Finally, Sect. 5 concludes the paper and surveys related works.

2 *AbC* in a Nutshell

In this section we briefly introduce the *AbC* calculus by means of a running example. We give *an intuition* of how to model a distributed variant of the well known *Graph Colouring Problem* [14] using *AbC* constructs. We render the

problem as a typical CAS scenario where a collective of agents, executing the same code, collaborate to achieve a system-level goal without any centralised control. The presentation is intended to be intuitive and full details concerning the example, the syntax, and the semantics of AbC can be found in [1, 3].

The problem consists of assigning a *colour* (an integer) to each vertex in a graph while avoiding that two neighbours get the same colour. The algorithm consists of a sequence of rounds for colour selection. At the end of each round at least one vertex is assigned a colour. A vertex, with identity id , uses messages of the form (“*try*”, c, r, id) to inform its neighbours that at round r it wants to select colour c and messages of the form (“*done*”, c, r, id) to communicate that colour c has been definitely chosen at the end of round r . At the beginning of a round, each vertex selects a colour and sends a *try*-message to all of its neighbours N . A vertex also collects *try*-messages from its neighbours. The selected colour is assigned to a vertex only if it has the greatest id among those that have selected the same colour in that round. After the assignment, a *done*-message (associated with the current round) is sent to neighbours.

AbC Syntax. An AbC component (C), is either a process P associated with an *attribute environment* Γ (denoted by $\Gamma:P$) or the parallel composition $C_1\|C_2$ of components. The *attribute environment* Γ is a partial map from attribute identifiers $a \in \mathcal{A}$ to values $v \in \mathcal{V}$. Values can be numbers, strings, tuples, etc.

$$C ::= \Gamma:P \quad | \quad C_1\|C_2$$

Example (step 1/4): Each vertex, in the colouring scenario, can be modelled in AbC as a component of the form $C_i = \Gamma_i : P_C$. The overall system is the parallel composition of vertices (i.e., $C_1\|C_2\|, \dots, \|C_n$).

The attribute environment of a vertex Γ_i relies on the following attributes to control the behaviour of a vertex: The attribute “**round**” stores the current round while “**used**” is a set, registering the colours used by neighbours. The attribute “**counter**” counts the number of messages collected by a component while “**send**” is used to enable/disable forwarding of messages to neighbours. Attribute “**assigned**” indicates if a vertex is assigned a colour while “**colour**” is a colour proposal. Finally, attributes id and N are used to represent the vertex id and the set of *neighbours*, respectively. These attributes initially have the following values: $round = 0$, $used = \emptyset$, $send = tt$, and $assigned = ff$.

It should be noted that new values for these attributes can only be learnt by means of message exchange among vertices. \square

The behavior of an AbC process can be generated by the following grammar:

$$P ::= 0 \quad | \quad \alpha.P \quad | \quad [\tilde{a} := \tilde{E}]P \quad | \quad \langle \Pi \rangle P \quad | \quad P_1 + P_2 \quad | \quad P_1|P_2 \quad | \quad K$$

The process 0 denotes the inactive process; $\alpha.P$ denotes a process that executes action α and continues as P ; process $[\tilde{a} := \tilde{E}]P$ behaves as P given that its attribute environment is first updated by setting the value of each attribute in the sequence \tilde{a} to the evaluation of the corresponding expression in the sequence \tilde{E} . The attribute updates and the first move of P are atomic; $\langle \Pi \rangle P$ denotes an

Table 1. *AbC* communication rules

$$\begin{array}{c}
\frac{\Gamma : P \xrightarrow{\lambda} \Gamma' : P'}{\Gamma : P \xrightarrow{\lambda} \Gamma' : P'} \text{ ICOMP} \qquad \frac{\Gamma : P \xrightarrow{\widetilde{\Pi(\tilde{v})}} \Gamma : P}{\Gamma : P \xrightarrow{\Pi(\tilde{v})} \Gamma : P} \text{ FCOMP} \\
\frac{C_1 \xrightarrow{\widetilde{\Pi(\tilde{v})}} C'_1 \quad C_2 \xrightarrow{\Pi(\tilde{v})} C'_2}{C_1 \parallel C_2 \xrightarrow{\widetilde{\Pi(\tilde{v})}} C'_1 \parallel C'_2} \text{ COM} \qquad \frac{C_1 \xrightarrow{\Pi(\tilde{v})} C'_1 \quad C_2 \xrightarrow{\Pi(\tilde{v})} C'_2}{C_1 \parallel C_2 \xrightarrow{\Pi(\tilde{v})} C'_1 \parallel C'_2} \text{ SYNC}
\end{array}$$

awareness process, it blocks the execution of process P until the predicate Π evaluates to true; the processes $P_1 + P_2$, $P_1|P_2$, and K are standard for nondeterminism, parallel composition, and process definition respectively. The parallel operator “|” does not allow communication between P_1 and P_2 , they can only interleave while the parallel operator “||” at the component level allows communication between components. The expression `this.b` denotes the value of attribute \mathbf{b} in the current component.

Example (step 2/4): Process P_C , specifying the behaviour of a vertex is now defined as the parallel composition of these four processes: $P_C \triangleq F | T | D | A$.

Process F forwards *try*-messages to neighbours, T handles *try*-messages, D handles *done*-messages, and A is used for assigning a final colour. \square

The *AbC* communication actions ranged by α can be either $(\tilde{E})@II$ or $\Pi(\tilde{x})$. The construct $(\tilde{E})@II$ denotes an output action, it evaluates the sequence of expressions \tilde{E} under the local attribute environment and then sends the result to the components whose attributes satisfy the predicate II . Furthermore, $\Pi(\tilde{x})$ denotes an input action, it binds to sequence \tilde{x} the corresponding received values from components whose *communicated attributes* or values satisfy II .

Example (step 3/4): We further specify process F and a part of process T .

$$F \triangleq \langle \text{send} \wedge \neg \text{assigned} \rangle [\text{colour} := \min\{i \notin \text{this.used}\}, \text{send} := \text{ff}] \\
(\text{“try”}, \text{this.colour}, \text{this.round}, \text{this.id}) @ (\text{this.id} \in \mathbf{N}).F$$

$$T \triangleq [\text{counter} := \text{counter} + 1] \\
((x = \text{“try”}) \wedge (\text{this.id} > l) \wedge (\text{this.round} = z))(x, y, z, l).T + \dots$$

In process F , when the value of attribute `send` becomes true, a new colour is selected, `send` is turned off, and a message containing this colour and the current round is sent to all the vertices having `this.id` as neighbour. The new colour is the smallest colour that has not yet been selected by neighbours, that is $\min\{i \notin \text{this.used}\}$. The guard `¬assigned` is used to make sure that vertices with assigned colours do not take part in the colour selection anymore.

Process T receives messages of the form $(\text{“try”}, c, r, id)$. If $r = \text{this.round}$ then the received message has been originated by a vertex performing the same round of the algorithm. The condition `this.id > l` means that the sender has an *id* smaller than the *id* of the receiver. In this case, the message is ignored

(there is no conflict), simply the counter of collected messages (`this.counter`) is incremented. Other cases, not reported here, e.g., `this.id < l`, the received colour is recorded to check the presence of conflicts. \square

AbC Semantics. The main semantics rules of *AbC* are reported in Table 1. Rule ICOMP states that a component evolves with (send $\overline{II}(\tilde{v})$ or receive $II(\tilde{v})$, denoted by λ) if its internal behaviour, denoted by the relation \mapsto , allows it. Rule FCOMP states that a component can discard a message $II(\tilde{v})$ if its internal behaviour does not allow the reception of this message by generating the discarding label $\widetilde{II}(\tilde{v})$. Rule COM¹ states that if C_1 evolves to C'_1 by sending a message $\overline{II}(\tilde{v})$ then this message should be delivered to C_2 which evolves to C'_2 as a result. Note that C_2 can be also a parallel composition of different components. Thus, rule SYNC states that multiple components can be delivered the same message in a single transition.

The semantics of the parallel composition operator, in rules COM and SYNC in Table 1, abstracts from the underlying coordination infrastructure that mediates the interactions between components and thus the semantics assumes atomic message exchange. This implies that no component can evolve before the sent message is delivered to all components executing in parallel. Individual components are in charge of using or discarding incoming messages. Message transmission is non-blocking, but reception is not. For instance, a component can still send a message even if there is no receiver (i.e., all the target components discard the message); a receive operation can, instead, only take place through synchronisation with an available message. However, if we want to use the attribute-based paradigm to program the interactions of distributed applications, atomicity and synchrony are neither efficient nor applicable.

One solution is to rely on existing protocols for total-order broadcast to handle message exchange. However, these protocols are mostly centralised [9] or rely on consensus [20]. Clearly, centralised solutions have always scalability and efficiency problems. Furthermore, consensus approaches are not only inefficient [20] but also impossible in asynchronous systems in the presence of even a single component's failure [13]. They also assume that components know each other and can agree on a specific order. However, this contradicts the main design principles of the *AbC* calculus where anonymity and openness are crucial factors. Since *AbC* components are *agnostic* to the infrastructure, they cannot participate in establishing the total order. Thus, we need an infrastructure that guarantees the total order seamlessly and without involving the interacting components.

The focus of this paper, as we will see later, is on providing an *efficient distributed* coordination infrastructure that behaves in agreement with the parallel composition operator of *AbC*. Thus in Table 1, we only formalised the external behaviour of a component, i.e., its ability to send and receive. The following example shows how interactions are derived based on internal behaviour.

¹ For the sake of brevity, we omit the symmetric rule of COM.

Example (step 4/4): Consider the vertices C_1, C_2 , and C_3 where $\Gamma_2(\mathbb{N}) = \{3\}$, $\Gamma_3(\mathbb{N}) = \{1, 4\}$, $\Gamma_3(\text{id}) = 3$, and $\Gamma_3(\text{round}) = 5$. Now C_1 sent a try message:

$$\Gamma_1 : P_C \xrightarrow{\overline{(1 \in \mathbb{N})}(\text{“try”}, 3, 5, 1)} \overbrace{\Gamma_1[\text{colour} \leftarrow 3, \text{send} \leftarrow \text{ff}]}^{C'_1} : P'_C$$

We have that C_2 discards this message because $\Gamma_2 \not\models (1 \in \mathbb{N})$ while C_3 accepts the message ($\Gamma_3 \models (1 \in \mathbb{N})$ and the receiving predicate of process T is satisfied). The system evolves with rule COM as follows:

$$C_1 \| C_2 \| C_3 \xrightarrow{\overline{(1 \in \mathbb{N})}(\text{“try”}, 3, 5, 1)} C'_1 \| C_2 \| \Gamma_3[\text{counter} \leftarrow \text{counter} + 1] : P'_C[\text{“try”}/x, 3/y, 5/z, 1/l]$$

□

3 A Distributed Coordination Infrastructure

In this section, we consider three possible coordination infrastructures that we have also implemented² in Google Go. We will refer to them as *cluster*-based, *ring*-based, and *tree*-based. These infrastructures behave in agreement with the parallel composition operator of AbC . Our approach consists of labelling each message with an id that is uniquely identified at the infrastructure level. Components execute asynchronously while the semantics of the parallel composition operator is preserved by relying on the unique identities of exchanged messages. In essence, if a component wants to send a message, it sends a request to the infrastructure for a fresh id. The infrastructure replies back with a fresh id and then the component sends a data (the actual) message with the received id. A component receives a data message only when the difference between the incoming data message id and the id of the last received data message equals 1. Otherwise the data message is added to the component waiting queue until the condition is satisfied.

In what follows, we give a full formal account of the proposed infrastructures and also investigate the correctness of the tree-based one. The reason is that we want to avoid redundancy and also because the tree infrastructure is theoretically the most challenging one. Actually, the proofs of correctness for the other infrastructures are simple cases of the tree’s one. Moreover, as we will see in Sect. 4, the tree exhibits better performance characteristics.

Furthermore to provide compact semantics, we use the following definition of a *Configuration*. For the sake of clarity, we will postfix the configuration of a component, an infrastructure, and a server with the letter a , n , and s respectively.

Definition 1 (Configuration). *A configuration C , is a tuple $C = \langle c_1, \dots, c_n \rangle$ which is commutative. The symbol ‘...’ is formally regarded as a meta-variable ranging over unmentioned elements of the configuration. The explicit ‘...’ is*

² Go implementations: <https://github.com/giulio-garbi/goat>.

obligatory, and ensures that unmentioned elements of a configuration are never excluded, but they do not play any role in the current context. Different occurrences of ‘...’ in the same context stand for the same set of unmentioned elements.

We use the reduction relation $\rightsquigarrow \subseteq \text{CFG} \times \text{LAB} \times \text{CFG}$ to define the semantics of a configuration where CFG denotes the set of configurations, LAB denotes the set of reduction labels which can be a message m , a silent transition τ , or an empty label, and \rightsquigarrow^* denotes the transitive closure of \rightsquigarrow . Moreover, we will use the following notations:

- We have two kinds of messages, an AbC message ‘msg’ (i.e., $\Pi(\tilde{v})$) and an infrastructure message ‘m’; the latter can be of three different templates: (i) request {‘Q’, route, dest}, (ii) reply {‘R’, id, route, dest}, and (iii) data {‘D’, id, src, dest, msg}. The route field in a request or a reply message is a linked list containing the addresses of the nodes that the message traversed.
- The notation $\stackrel{?}{=}$ denotes a template matching.
- The notation $T[f]$ denotes the value of the element f in T .

Also the following operations will be used: $L.get()$ returns the element at the front of a list/queue, while $L \leftarrow m$ returns the list/queue resulting from adding m to the back of L , and $L \setminus x$ removes x from L and returns the rest.

3.1 Infrastructure Component

Now, we formally define a general infrastructure component and its external behaviour. In the following sections, we proceed by formally defining the proposed infrastructures and their behaviours.

Definition 2 (Infrastructure component). *An infrastructure component, a , is defined by the configuration: $a = \langle addr, nid, mid, on, \mathcal{W}, \mathcal{X}, G \rangle$ where $addr$ refers to its address, nid (initially 0) refers to the id of the next data message to be received, mid (initially -1) refers to the id of the most recent reply, on (initially 0) indicates whether a request message can be sent. \mathcal{W} is a priority waiting queue where the top of \mathcal{W} is the data message with the least id, and \mathcal{X} refers to the address of the parent server. Furthermore, G ranges over $\Gamma:P$ and $[\Gamma:P]$ where $[\Gamma:P]$ indicates an AbC component in an intermediate state.*

The intermediate state, in Definition 2, is important to allow co-located processes (i.e., $[\Gamma:P_1P_2]$ where P_1 is waiting an id to send and P_2 is willing to receive) to interleave their behaviours without compromising the semantics.

The semantics of an infrastructure component is reported in Table 2. Rule OUT states that if the AbC component $\Gamma:P$ encapsulated inside an infrastructure component is able to send a message $\Gamma:P \xrightarrow{\Pi(\tilde{v})} \Gamma':P'$, the flag on is set to 1 and $\Gamma:P$ goes into an intermediate state $[\Gamma:P]$. Rule MED states that an intermediate state component can only receive a message $\Pi(\tilde{v})$ if it was able to receive it before the intermediate state. Rule REQ states that a component

Table 2. The semantics of a component

$$\begin{array}{c}
\frac{\Gamma:P \xrightarrow{\overline{\Pi}(\tilde{v})} \Gamma':P'}{\langle on, \Gamma:P, \dots \rangle_a \xrightarrow{\tau} \langle 1, [\Gamma:P], \dots \rangle_a} \text{ OUT} \quad \frac{\Gamma:P \xrightarrow{\Pi(\tilde{v})} \Gamma':P'}{[\Gamma:P] \xrightarrow{\Pi(\tilde{v})} [\Gamma':P']} \text{ MED} \\
\frac{on == 1}{\langle addr, on, \mathcal{X}, \dots \rangle_a \xrightarrow{\{\text{'Q'}, \{addr\}, \mathcal{X}\}} \langle addr, 0, \mathcal{X}, \dots \rangle_a} \text{ REQ} \\
\langle addr, mid, \dots \rangle_a \xrightarrow{\{\text{'R'}, id, \{\}, addr\}} \langle addr, id, \dots \rangle_a \text{ RCVR} \\
\frac{nid == mid \quad \Gamma:P \xrightarrow{\overline{\Pi}(\tilde{v})} \Gamma':P'}{\langle addr, nid, mid, [\Gamma:P], \mathcal{X}, \dots \rangle_a \xrightarrow{\{\text{'D'}, mid, addr, \mathcal{X}, \Pi(\tilde{v})\}} \langle addr, nid + 1, -1, \Gamma':P', \mathcal{X}, \dots \rangle_a} \text{ SND} \\
\frac{m \stackrel{?}{=} \{\text{'D'}, id, \mathcal{X}, addr, msg\} \quad id \geq nid}{\langle addr, nid, \mathcal{W}, \mathcal{X}, \dots \rangle_a \xrightarrow{m} \langle addr, nid, \mathcal{W} \leftarrow m, \mathcal{X}, \dots \rangle_a} \text{ RCVD} \\
\frac{m[id] == nid \quad G \xrightarrow{m[msg]} G'}{\langle nid, G, m :: \mathcal{W}', \dots \rangle_a \xrightarrow{\tau} \langle nid + 1, G', \mathcal{W}', \dots \rangle_a} \text{ HND}
\end{array}$$

sends a request, to the parent server, only if $on == 1$. In this case, it adds its address to the *route* of the message and resets on to 0. Rule RCVR states that a component receives a reply if the destination field of the reply matches its address; after that mid gets the value of the id received in the reply. Rule SND states that a component $\Gamma:P$ can send a message $\overline{\Pi}(\tilde{v})$ and evolves to $\Gamma':P'$ only if $nid == mid$; this implies that a fresh id is received ($mid \neq -1$) and all messages with $m[id] < mid$ have been already received. By doing so, an infrastructure data message, with msg field equals to $\Pi(\tilde{v})$, is sent, nid is incremented, and mid is reset. Rule RCVD states that a component receives a data message from the infrastructure if $m[id] \geq nid$; this is important to avoid duplicate messages. The message is then added to the priority queue, \mathcal{W} . Finally, rule HND states that when the id of the message on top of \mathcal{W} matches nid , component G is allowed to receive that message; by doing so, nid is incremented and m is removed.

3.2 Cluster-Based Infrastructure

We consider a set of server nodes, sharing a counter for sequencing messages and one FIFO queue to store messages sent by components. Cluster nodes can have exclusive locks on both the cluster's counter and the queue. Components register directly to the cluster and send messages to be added to the FIFO queue. When a server node retrieves a request from the cluster queue, it replies to the requester with the value of the cluster counter. By doing so, the cluster counter

Table 3. The Cluster semantics

$$\begin{array}{c}
\frac{a \xrightarrow{m} a' \quad m[\text{dest}] == \text{addr}}{\langle \text{addr}, \{a\} \cup \mathcal{A}', \mathcal{I}, \dots \rangle_n \rightsquigarrow \langle \text{addr}, \{a'\} \cup \mathcal{A}', \mathcal{I} \leftarrow m, \dots \rangle_n} \text{QIN} \\
\\
\frac{s \xrightarrow{m} s'}{\langle \{s\} \cup \mathcal{S}', m :: \mathcal{I}', \dots \rangle_n \rightsquigarrow \langle \{s'\} \cup \mathcal{S}', \mathcal{I}', \dots \rangle_n} \text{QOUT} \\
\\
\frac{a \xrightarrow{\tau} a'}{\langle \{a\} \cup \mathcal{A}', \dots \rangle_n \rightsquigarrow \langle \{a'\} \cup \mathcal{A}' \dots \rangle_n} \text{A} \\
\\
\frac{\begin{array}{c} \{^{\text{D}}, \text{id}, \text{addr}', \text{addr}, \text{msg}\} \\ s \rightsquigarrow s' \end{array} \quad \begin{array}{c} \{^{\text{D}}, \text{id}, \text{addr}', \text{addr}, \text{msg}\} \\ a \rightsquigarrow a' \end{array} \quad a[\text{addr}] == \text{addr}}{\langle \{s\} \cup \mathcal{S}', \{a\} \cup \mathcal{A}', \dots \rangle_n \rightsquigarrow \langle \{s'\} \cup \mathcal{S}', \{a'\} \cup \mathcal{A}', \dots \rangle_n} \text{DMSG} \\
\\
\frac{\begin{array}{c} \{^{\text{R}}, \cdot, \{\}, \text{addr}\} \\ s \rightsquigarrow s' \end{array} \quad \begin{array}{c} \{^{\text{R}}, \text{ctr}, \{\}, \text{addr}\} \\ a \rightsquigarrow a' \end{array}}{\langle \{s\} \cup \mathcal{S}', \{a\} \cup \mathcal{A}', \text{ctr}, \dots \rangle_n \rightsquigarrow \langle \{s'\} \cup \mathcal{S}', \{a'\} \cup \mathcal{A}', \text{ctr} + 1, \dots \rangle_n} \text{RMSG}
\end{array}$$

is incremented. If a server retrieves a data message, it forwards the message to all components in the cluster except for the sender.

Definition 3 (Cluster node). A server node, s , is defined by the configuration $s = \langle \text{addr}, \mathcal{A}, \mathcal{M}, \mathcal{I} \rangle$ where addr is its address, \mathcal{A} is a set containing the addresses of all cluster components, \mathcal{M} is a multicast set (initially $\mathcal{M} = \mathcal{A}$). Finally, \mathcal{I} is a FIFO input queue.

Definition 4 (Cluster infrastructure). A cluster, \mathcal{N} , is defined by the configuration $\mathcal{N} = \langle \text{addr}, \text{ctr}, \mathcal{S}, \mathcal{A}, \mathcal{I} \rangle$ where ctr is a counter to generate fresh ids, initially the value of ctr equals 0, \mathcal{S} is a set containing the addresses of the infrastructure server nodes, and the rest is defined as before.

We start by defining the overall infrastructure semantics and then we zoom in and we define the semantics of individual servers. The cluster semantics is reported in Table 3. Rule QIN states that a component sends a message and the cluster adds it to its input queue. Rule QOUT states that the cluster evolves when a server gets a message from the input queue of the cluster. Rule “A” states that the cluster evolves when one of its components evolves independently. Rule DMSG states that the cluster evolves when a server can forward a data message to a component in the cluster. Rule RMSG states that the cluster evolves when a node sends a reply message to a component. The reply is labeled with the current value of the the cluster counter and after that the counter is incremented.

The semantics of a cluster node is reported in Table 4. Rule IN states that a node gets a message and adds it to its input queue. Rule REPLY states that if a node gets a request message from its input queue, it sends a reply to the requester by getting and removing its address from the route of the message. Rule DFWD states that if a node gets a data message from its input queue, it

Table 4. Cluster node semantics
$$\begin{array}{c}
\frac{}{\langle \mathcal{I}, \dots \rangle_s \xrightarrow{m} \langle \mathcal{I} \leftarrow m, \dots \rangle_s} \text{IN} \qquad \frac{m \stackrel{?}{=} \{ 'Q', \{ addr' \}, addr \}}{\langle m :: \mathcal{I}', \dots \rangle_n \xrightarrow{\{ 'R', \{ \}, addr' \}} \langle \mathcal{I}', \dots \rangle_n} \text{REPLY} \\
\frac{|\mathcal{M}| > 1 \quad m \stackrel{?}{=} \{ 'D', id, addr', addr'', msg \} \quad \mathcal{M}' = \mathcal{M} \setminus addr' \quad x = \mathcal{M}'.get()}{\langle addr, \mathcal{A}, \mathcal{M}, m :: \mathcal{I}' \rangle_s \xrightarrow{\{ 'D', id, addr, x, msg \}} \langle addr, \mathcal{A}, \mathcal{M}' \setminus x, m :: \mathcal{I}' \rangle_s} \text{DFWD} \\
\frac{|\mathcal{M}| = 1 \quad m \stackrel{?}{=} \{ 'D', id, addr', addr, msg \} \quad x = \mathcal{M}.get()}{\langle addr, \mathcal{A}, \mathcal{M}, m :: \mathcal{I}' \rangle_s \xrightarrow{\{ 'D', id, addr, x, msg \}} \langle addr, \mathcal{A}, \mathcal{A}, \mathcal{I}' \rangle_s} \text{EFWD}
\end{array}$$

forwards the message to all components in the cluster one by one except for the sender ($addr'$). Notice that this rule can be applied many times as long as the multicast set \mathcal{M} contains more than one element, i.e., $|\mathcal{M}| > 1$. Once \mathcal{M} has only one element, rule EFWD is applied to forward the message to the last address in \mathcal{M} , resets the multicast set to its initial value, and the message is removed.

3.3 Ring-Based Infrastructure

We consider a set of server nodes, organised in a logical ring and sharing a counter for sequencing messages coming from components. Each node manages a group of components and can have exclusive locks to the ring counter. When a request message arrives to a node from one of its components, the node acquires a lock on the ring counter, copies its current value, releases it after incrementing it by 1, and finally sends a reply, carrying a fresh id, to the requester. Data messages are directly added to the node's waiting queue; and will be only forwarded to the node's components and to the neighbour node when all previous messages (i.e., with a smaller id) have been received.

Definition 5 (Ring node). A server node, s , is defined by the configuration $s = \langle addr, nid, \mathcal{X}, \mathcal{D}, \mathcal{M}, \mathcal{I}, \mathcal{W} \rangle$ where \mathcal{X} is its neighbour's address, \mathcal{D} is a set containing the addresses of components connected to this server node and also the neighbour's address \mathcal{X} , and \mathcal{M} initially equals \mathcal{D} . The rest is defined as before.

Definition 6 (Ring infrastructure). A ring, \mathcal{N} , is defined by the configuration $\mathcal{N} = \langle \mathcal{S}, \mathcal{A}, ctr \rangle$. We have that:

- $\forall s \in \mathcal{N}[\mathcal{S}] : s[\mathcal{X}] \neq \perp \wedge s[\mathcal{X}] \in \mathcal{N}[\mathcal{S}]$.
- $\forall s_1, s_2 \in \mathcal{N}[\mathcal{S}] : s_1[\mathcal{X}] = s_2[\mathcal{X}]$ implies $s_1 = s_2$.

The semantics rules of a ring infrastructure are reported in Table 5. The rules (S \leftrightarrow S) and (S \leftrightarrow A) state that a ring evolves when a message m is exchanged either between two of its servers (s_1 and s_2) or between a server and a component respectively. The latter rule concerns only request and data

Table 5. Ring infrastructure semantics

$$\begin{array}{c}
\frac{s_1 \xrightarrow{m} s'_1 \quad s_2 \xrightarrow{m} s'_2}{\langle \{s_1, s_2\} \cup \mathcal{S}', \mathcal{A}, \dots \rangle_n \rightsquigarrow \langle \{s'_1, s'_2\} \cup \mathcal{S}', \mathcal{A}, \dots \rangle_n} \text{S} \leftrightarrow \text{S} \\
\frac{s \xrightarrow{\tau} s'}{\langle \{s\} \cup \mathcal{S}', \mathcal{A}, \dots \rangle_n \rightsquigarrow \langle \{s'\} \cup \mathcal{S}', \mathcal{A}, \dots \rangle_n} \text{S} \\
\frac{s \xrightarrow{m} s' \quad a \xrightarrow{m} a' \quad m \stackrel{?}{\neq} \{ 'R', \{\}, \text{addr} \}}{\langle \{s\} \cup \mathcal{S}', \{a\} \cup \mathcal{A}', \dots \rangle_n \rightsquigarrow \langle \{s'\} \cup \mathcal{S}', \{a'\} \cup \mathcal{A}', \dots \rangle_n} \text{S} \leftrightarrow \text{A} \\
\frac{a \xrightarrow{\tau} a'}{\langle \mathcal{S}, \{a\} \cup \mathcal{A}', \dots \rangle_n \rightsquigarrow \langle \mathcal{S}, \{a'\} \cup \mathcal{A}', \dots \rangle_n} \text{A} \\
\frac{s \overset{\{ 'R', \{\}, \text{addr} \}}{\rightsquigarrow} s' \quad a \overset{\{ 'R', \text{ctr}, \{\}, \text{addr} \}}{\rightsquigarrow} a'}{\langle \{s\} \cup \mathcal{S}', \{a\} \cup \mathcal{A}', \text{ctr} \rangle_n \rightsquigarrow \langle \{s'\} \cup \mathcal{S}', \{a'\} \cup \mathcal{A}', \text{ctr} + 1 \rangle_n} \text{RMSG}
\end{array}$$

messages. Furthermore, the rules (S) and (A) state that a ring evolves when one of its servers or one of its connected components evolves independently. Finally, rule RMSG states that a ring evolves also when a reply is exchanged between a server node and a component, but in this case the counter of the ring is increased.

The semantics rules of a ring node are reported in Tabel 6. Rule IN states that a node receives a message m and adds it to its input queue ($\mathcal{I} \leftarrow m$) if the destination field of m matches its own address addr . Rule REPLY states that if a node gets a request message from its input queue $m :: \mathcal{I}$, it sends a reply, to the requester. Rule WIN states that if a node gets a data message from its input queue, it adds the message to its waiting queue \mathcal{W} only if $m[\text{id}] \geq \text{nid}$ otherwise the message is discarded as stated by rule DISCARD. This is important to avoid duplicates. Furthermore, rule DFWD states that when the id of the message on top of \mathcal{W} matches nid (i.e., $m[\text{id}] == \text{nid}$), the server starts forwarding m to its children one by one except for the sender. Notice that this rule can be applied many times as long as the multicast set \mathcal{M} contains more than one element, i.e., $|\mathcal{M}| > 1$. Once \mathcal{M} has only one element, rule EFWD is applied to forward the message to the last address in \mathcal{M} . As a result, nid is incremented, m is removed from \mathcal{W} , and the multicast set \mathcal{M} is reset to its initial value.

3.4 A Tree-Based Infrastructure

We consider a set of servers, organised in a logical tree. A component can be connected to one server (its parent) in the tree and can interact with others in any part of the tree by only dealing with its parent. When a component wants to send a message, it asks for a fresh id from its parent. If the parent is the root of the tree, it replies with a fresh id, otherwise it forwards the message to its own parent in the tree. Only the root of the tree can sequence messages.

Table 6. Ring Node Semantics

$$\begin{array}{c}
\frac{m[\text{dest}] == \text{addr}}{\langle \text{addr}, \mathcal{I}, \dots \rangle_s \xrightarrow{m} \langle \text{addr}, \mathcal{I} \leftarrow m, \dots \rangle_s} \text{IN} \\
\\
\frac{m \stackrel{?}{=} \{\text{'Q'}, \{\text{addr}'\}, \text{addr}\}}{\langle \text{addr}, m :: \mathcal{I}', \dots \rangle_s \xrightarrow{\{\text{'R'}, \{\}, \text{addr}'\}} \langle \text{addr}, \mathcal{I}', \dots \rangle_s} \text{REPLY} \\
\\
\frac{m \stackrel{?}{=} \{\text{'D'}, \text{id}, \text{addr}', \text{addr}, \text{msg}\} \quad \text{id} \geq \text{nid}}{\langle \text{addr}, \text{nid}, \mathcal{W}, m :: \mathcal{I}', \dots \rangle_s \xrightarrow{\tau} \langle \text{addr}, \text{nid}, \mathcal{W} \leftarrow m, \mathcal{I}', \dots \rangle_s} \text{WIN} \\
\\
\frac{m \stackrel{?}{=} \{\text{'D'}, \text{id}, \text{addr}', \text{addr}, \text{msg}\} \quad \text{id} < \text{nid}}{\langle \text{addr}, \text{nid}, \mathcal{W}, m :: \mathcal{I}', \dots \rangle_s \xrightarrow{\tau} \langle \text{addr}, \text{nid}, \mathcal{W}, \mathcal{I}', \dots \rangle_s} \text{DISCARD} \\
\\
\frac{|\mathcal{M}| > 1 \quad \frac{m[\text{id}] == \text{nid} \quad \mathcal{M}' = \mathcal{M} \setminus \{\text{addr}' \quad x = \mathcal{M}'.\text{get}()\}}{m \stackrel{?}{=} \{\text{'D'}, \text{id}, \text{addr}', \text{addr}, \text{msg}\}}}{\langle \text{addr}, \text{nid}, \mathcal{D}, \mathcal{M}, m :: \mathcal{W}', \dots \rangle_s \xrightarrow{\{\text{'D'}, \text{id}, \text{addr}, x, \text{msg}\}} \langle \text{addr}, \text{nid}, \mathcal{D}, \mathcal{M}' \setminus x, m :: \mathcal{W}', \dots \rangle_s} \text{DFWD} \\
\\
\frac{|\mathcal{M}| = 1 \quad \frac{m[\text{id}] == \text{nid} \quad m \stackrel{?}{=} \{\text{'D'}, \text{id}, \text{addr}', \text{addr}, \text{msg}\} \quad x = \mathcal{M}.\text{get}()}{\langle \text{addr}, \text{nid}, \mathcal{D}, \mathcal{M}, m :: \mathcal{W}', \dots \rangle_s \xrightarrow{\{\text{'D'}, \text{id}, \text{addr}, x, \text{msg}\}} \langle \text{addr}, \text{nid} + 1, \mathcal{D}, \mathcal{D}, \mathcal{W}', \dots \rangle_s} \text{EFWD}}
\end{array}$$

Definition 7 (Tree server). A tree server, s , is defined by the configuration: $s = \langle \text{addr}, \text{ctr}, \text{nid}, \mathcal{D}, \mathcal{M}, \mathcal{I}, \mathcal{W}, \mathcal{X} \rangle$ where \mathcal{D} is a set containing the addresses of the server's children which include connected components and servers, \mathcal{M} is a multicast set (initially $\mathcal{M} = \mathcal{D}$). The rest are defined as before.

Definition 8 (Tree infrastructure). A tree infrastructure, \mathcal{N} , is defined by the configuration: $\mathcal{N} = \langle \mathcal{S}, \mathcal{A} \rangle$ where \mathcal{S} denotes the set of servers and \mathcal{A} denotes the set of connected components such that:

- $\forall s_1, s_2 \in \mathcal{S}$, we say that s_1 is a direct child of s_2 , written $s_1 \prec s_2$, if and only if $s_1[\mathcal{X}] = s_2[\text{addr}]$; \prec^+ denotes the transitive closure of \prec .
- $\forall s \in \mathcal{S}$, we have that $s \not\prec^+ s$.
- The root: $\exists s \in \mathcal{S}$ such that for any $s' \in (\mathcal{S} \setminus \{s\})$, $s' \prec^+ s$ and we have that:
 - $s'[\text{nid}] \leq s[\text{ctr}]$.
 - For any message $m \in s'[\mathcal{W}]$ we have that $m[\text{id}] \leq s[\text{ctr}]$.
- A root is unique: if $s, s' \in \mathcal{S}$ and $s[\mathcal{X}] = s'[\mathcal{X}] = \perp$ then we have that $s = s'$.
- $\forall s \in \mathcal{S}$ and for each message $m \in s[\mathcal{W}]$, we have that $m[\text{id}] \geq s[\text{nid}]$.

The semantics rules of a tree infrastructure are reported in Table 7. The rules (S \leftrightarrow S) and (S \leftrightarrow A) state that a tree evolves when a message m is exchanged either between two of its servers (s_1 and s_2) or between a server and a component respectively. Furthermore, the rules (S) and (A) state that a tree evolves when one of its servers or one of its connected components evolves independently.

Table 7. Tree infrastructure semantics

$$\begin{array}{c}
\frac{s_1 \xrightarrow{m} s'_1 \quad s_2 \xrightarrow{m} s'_2}{\langle \{s_1, s_2\} \cup \mathcal{S}', \mathcal{A} \rangle_n \rightsquigarrow \langle \{s'_1, s'_2\} \cup \mathcal{S}', \mathcal{A} \rangle_n} \text{S} \leftrightarrow \text{S} \quad \frac{s \xrightarrow{\tau} s'}{\langle \{s\} \cup \mathcal{S}', \mathcal{A} \rangle_n \rightsquigarrow \langle \{s'\} \cup \mathcal{S}', \mathcal{A} \rangle_n} \text{S} \\
\frac{s \xrightarrow{m} s' \quad a \xrightarrow{m} a'}{\langle \{s\} \cup \mathcal{S}', \{a\} \cup \mathcal{A}' \rangle_n \rightsquigarrow \langle \{s'\} \cup \mathcal{S}', \{a'\} \cup \mathcal{A}' \rangle_n} \text{S} \leftrightarrow \text{A} \quad \frac{a \xrightarrow{\tau} a'}{\langle \mathcal{S}, \{a\} \cup \mathcal{A}' \rangle_n \rightsquigarrow \langle \mathcal{S}, \{a'\} \cup \mathcal{A}' \rangle_n} \text{A}
\end{array}$$

The semantics rules of a tree server are defined by the rules in Table 8. Rule IN states that a server receives a message m and adds it to the back of its input queue ($\mathcal{I} \leftarrow m$) if the destination field of m matches its own address $addr$. Rule REPLY states that if a root server gets a request from the front of its input queue $m :: \mathcal{T}'$, it sends a reply to the requester by getting its address from the route of the message $x = route.get()$. The id of the reply is assigned the value of the root's counter ctr . By doing so, the counter is incremented. On the other hand, a non-root server adds its address to the message's route and forwards it to its parent as stated by rule QFWD. Rule RFWD instead is used for forwarding reply messages. Rule WIN states that if a server gets a data s message from its input queue \mathcal{I} and it is the root or its parent is the source of the message (i.e., $\mathcal{X} == addr' \vee \mathcal{X} == \perp$), the server evolves silently and the message is added to its waiting queue. If the condition ($\mathcal{X} == addr' \vee \mathcal{X} == \perp$) does not hold, the message is also forwarded to the parent as stated by rule WNXT. Furthermore, rule DFWD states that when the id of the message on top of \mathcal{W} matches nid (i.e., $m[id] == nid$), the server starts forwarding m to its children one by one except for the sender. Notice that this rule can be applied many times as long as the multicast set \mathcal{M} contains more than one element, i.e., $|\mathcal{M}| > 1$. Once \mathcal{M} has only one element, rule EFWD is applied to forward the message to the last address in \mathcal{M} . As a result, nid is incremented, m is removed from \mathcal{W} , and the multicast set \mathcal{M} is reset to its initial value.

Correctness. Since there is a single sequencer in the tree, i.e., the root, two messages can never have the same id. We only need the following propositions to ensure that the tree behaves in agreement with the *AbC* parallel composition operator. In essence, Proposition 1, ensures that if any component in the tree sends a request for a fresh id, it will get it. Proposition 3, ensures that any two components in the tree with different nid will converge to the same one. However, to prove Proposition 3, we need to prove Lemma 1 and Proposition 2 which guarantee the same results among tree' servers. This implies that messages are delivered to all components. Proposition 4 instead ensures that no message stays in the waiting queue indefinitely. Due to space limitations all proofs are omitted.

Proposition 1. *For any component, with address $addr$ and a parent \mathcal{X} , connected to a tree infrastructure \mathcal{N} , we have that: if $\langle addr, on, \mathcal{X}, \dots \rangle_a$*

Table 8. Tree server semantics

$$\begin{array}{c}
\frac{m[\text{dest}] == \text{addr}}{\langle \text{addr}, \mathcal{I}, \dots \rangle_s \xrightarrow{m} \langle \text{addr}, \mathcal{I} \leftarrow m, \dots \rangle_s} \text{IN} \\
\\
\frac{m \stackrel{?}{=} \{ 'Q', \text{route}, \text{addr} \} \quad \mathcal{X} == \perp \quad x = \text{route.get}()}{\langle \text{addr}, \text{ctr}, \mathcal{X}, m :: \mathcal{I}', \dots \rangle_s \xrightarrow{\{ 'R', \text{ctr}, \text{route} \setminus x, x \}} \langle \text{addr}, \text{ctr} + 1, \mathcal{X}, \mathcal{I}', \dots \rangle_s} \text{REPLY} \\
\\
\frac{m \stackrel{?}{=} \{ 'Q', \text{route}, \text{addr} \} \quad \mathcal{X} \neq \perp \quad \text{route}' = \text{route.add}(\text{addr})}{\langle \text{addr}, \mathcal{X}, m :: \mathcal{I}', \dots \rangle_s \xrightarrow{\{ 'Q', \text{route}', \mathcal{X} \}} \langle \text{addr}, \mathcal{X}, \mathcal{I}', \dots \rangle_s} \text{QFWD} \\
\\
\frac{m \stackrel{?}{=} \{ 'R', \text{id}, \text{route}, \text{addr} \} \quad x = \text{route.get}()}{\langle \text{addr}, m :: \mathcal{I}', \dots \rangle_s \xrightarrow{\{ 'R', \text{id}, \text{route} \setminus x, x \}} \langle \text{addr}, \mathcal{I}', \dots \rangle_s} \text{RFWD} \\
\\
\frac{m \stackrel{?}{=} \{ 'D', \text{id}, \text{addr}', \text{addr}, \text{msg} \} \quad (\mathcal{X} == \text{addr}' \vee \mathcal{X} == \perp)}{\langle \text{addr}, \mathcal{X}, \mathcal{W}, m :: \mathcal{I}', \dots \rangle_s \xrightarrow{\tau} \langle \text{addr}, \mathcal{X}, \mathcal{W} \leftarrow m, \mathcal{I}', \dots \rangle_s} \text{WIN} \\
\\
\frac{m \stackrel{?}{=} \{ 'D', \text{id}, \text{addr}', \text{addr}, \text{msg} \} \quad (\mathcal{X} \neq \text{addr}' \wedge \mathcal{X} \neq \perp)}{\langle \text{addr}, \mathcal{X}, \mathcal{W}, m :: \mathcal{I}', \dots \rangle_s \xrightarrow{\{ 'D', \text{id}, \text{addr}, \mathcal{X}, \text{msg} \}} \langle \text{addr}, \mathcal{X}, \mathcal{W} \leftarrow m, \mathcal{I}', \dots \rangle_s} \text{WNXT} \\
\\
\frac{|\mathcal{M}| > 1 \quad m[\text{id}] == \text{nid} \quad \mathcal{M}' = \mathcal{M} \setminus \text{addr}' \quad x = \mathcal{M}'.\text{get}()}{\langle \text{addr}, \text{nid}, \mathcal{D}, \mathcal{M}, m :: \mathcal{W}', \dots \rangle_s \xrightarrow{\{ 'D', \text{id}, \text{addr}, x, \text{msg} \}} \langle \text{addr}, \text{nid}, \mathcal{D}, \mathcal{M}' \setminus x, m :: \mathcal{W}', \dots \rangle_s} \text{DFWD} \\
\\
\frac{|\mathcal{M}| = 1 \quad m[\text{id}] == \text{nid} \quad m \stackrel{?}{=} \{ 'D', \text{id}, \text{addr}', \text{addr}, \text{msg} \} \quad x = \mathcal{M}.\text{get}()}{\langle \text{addr}, \text{nid}, \mathcal{D}, \mathcal{M}, m :: \mathcal{W}', \dots \rangle_s \xrightarrow{\{ 'D', \text{id}, \text{addr}, x, \text{msg} \}} \langle \text{addr}, \text{nid} + 1, \mathcal{D}, \mathcal{D}, \mathcal{W}', \dots \rangle_s} \text{EFWD}
\end{array}$$

$$\begin{array}{c}
\begin{array}{c} \{ 'Q', \{ \text{addr} \}, \mathcal{X} \} \\ \xrightarrow{\hspace{1cm}} \end{array} \langle \text{addr}, 0, \mathcal{X}, \dots \rangle_a \text{ then } \mathcal{N} \rightsquigarrow^* \mathcal{N}' \text{ and } \langle \text{addr}, \text{mid}, \dots \rangle_a \\
\begin{array}{c} \{ 'R', \text{id}, \{ \}, \text{addr} \} \\ \xrightarrow{\hspace{1cm}} \end{array} \langle \text{addr}, \text{id}, \dots \rangle_a.
\end{array}$$

Lemma 1. For every two tree nodes s_1 and s_2 and a tree-based infrastructure \mathcal{N} such that $s_1, s_2 \in \mathcal{N}[\mathcal{S}]$, we have that:

- If $s_1 \prec s_2 \wedge s_1[\text{nid}] < s_2[\text{nid}]$ then $\mathcal{N} \rightsquigarrow^* \mathcal{N}'$ and $s_1[\text{nid}] = s_2[\text{nid}]$.
- If $s_2 \prec s_1 \wedge s_1[\text{nid}] < s_2[\text{nid}]$ then $\mathcal{N} \rightsquigarrow^* \mathcal{N}'$ and $s_1[\text{nid}] = s_2[\text{nid}]$.

Proposition 2. Let s_1 and s_2 be two tree nodes and \mathcal{N} be a tree-based infrastructure, $\forall s_1, s_2 \in \mathcal{N}[\mathcal{S}] \wedge s_1[\text{nid}] < s_2[\text{nid}]$, we have that $\mathcal{N} \rightsquigarrow^* \mathcal{N}'$ and $s_1[\text{nid}] = s_2[\text{nid}]$.

Proposition 3. *Given any two components a_1 and a_2 in a tree infrastructure \mathcal{N} such that $a_1[nid] < a_2[nid]$, we have that $\mathcal{N} \rightsquigarrow^* \mathcal{N}'$ and $a_1[nid] = a_2[nid]$.*

Proposition 4. *Given a tree infrastructure $\mathcal{N} = \langle \mathcal{S}, \mathcal{A} \rangle$, for any $c \in \mathcal{S} \cup \mathcal{A}$ where $c[\mathcal{W}] = m :: \mathcal{W}'$, we have that $\mathcal{N} \rightsquigarrow^* \mathcal{N}'$ and $c[\mathcal{W}] = \mathcal{W}' \# \mathcal{W}''$ where $\#$ returns a priority queue composed by the sub queues \mathcal{W}' and \mathcal{W}'' .*

4 Performance Evaluation

We compare the above mentioned infrastructures by modeling them in terms of a Continuous Time Markov Process [17]. The state of a process represents possible infrastructure configurations, while the transitions (that are selected probabilistically) are associated with events on messages. We can consider three types of events: a new message *sent* by a component; a message *transmitted* from a node to another in the infrastructure; a message locally *handled* by a node (i.e. removed from an input/waiting queue). Each event is associated with a *rate* that is the parameter of the *exponentially distributed* random variable governing the *event duration*. We developed a simulator³ for performance evaluation.

To perform the simulation we need to fix three parameters: the *component sending rate* λ_s ; the *infrastructure transmission rate* λ_t ; and the *handling rate* λ_h . In all experiments, we fix the following values: $\lambda_s = 1.0$, $\lambda_t = 15.0$, and $\lambda_h = 1000.0$ and rely on kinetic Monte Carlo simulation [19]. The infrastructure configurations are defined as follows:

- $C[x, y]$, indicates a *cluster* with x nodes and y components;
- $R[x, y]$ indicates a *ring* with x nodes each of which manages y components;
- $T[x, y, z]$ indicates a *tree* with x levels. Each node (but the leaves) has $y + z$ children: y nodes and z components. A leaf node has z components.

We consider two scenarios: (1) *Data Providers* (DP): In this scenario only a fraction of components sends data messages that they, for example, acquire via sensors in the environment where they operate. An example could be a *Traffic Control System* where *data providers* are devices located in the city and *data receivers* are the vehicles traveling in the area; (2) *communication intensive* (CI): This scenario is used to estimate the performance when all components send messages continuously at a fixed rate so that we can evaluate situations of overloaded infrastructures. The former scenario is more realistic for CAS.

We consider two measures: the average delivery time and the average message time gap. The first measure indicates the time needed for a message to reach all components, while the latter indicates the interval between two different messages received by a single component (i.e., an indication of throughput).

³ The simulator: <https://bitbucket.org/Lazkany/abcsimulator>.

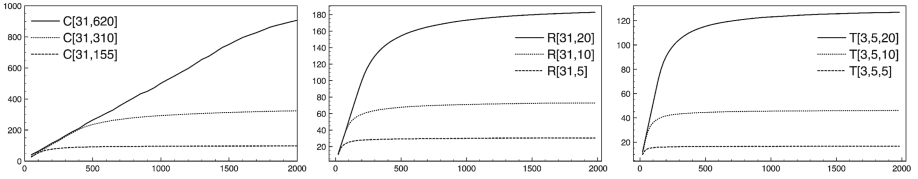


Fig. 1. DP scenario: Avg. Delivery Time for Cluster, Ring, and Tree.

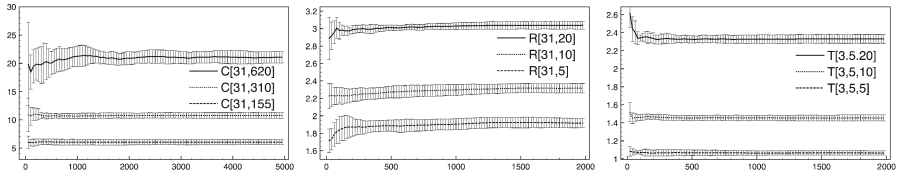


Fig. 2. DP scenario: Avg. Message Time Gap for Cluster, Ring, and Tree.

Data Provider Scenario (DP). We consider configurations with 31 server nodes 155, 310, or 620 components and assume that only 10% of the components is sending data. The average delivery time is reported in Fig. 1 while the average message time gap (with confidence intervals) is reported in Fig. 2. The tree structure offers the best performance while the cluster one is the worst. When the cluster reaches an equilibrium (at time ~ 2000), ~ 90 time units are needed to deliver a message to 155 components while the ring and the tree need only ~ 25 and ~ 10 time units, respectively. The reason is that in the cluster all server nodes share the same input queue while in the tree and the ring each server node has its own queue. We can also observe that the performance of the ring in this scenario is close to the one of the tree. Moreover, in the cluster, the performance degrades when the number of components increases. This does not happen for the tree and the ring. Finally, we can observe that messages are delivered more frequently in the ring (~ 1.9 time units) and the tree (~ 1.1 time units) than in the cluster (~ 5.5 time units) as reported in Fig. 2.

Communication Intensive Scenario (CI). We consider infrastructures composed by 155 components that continuously send messages to all the others. Simulations are performed by considering the following configurations:

- Cluster-based infrastructure: $C[10, 155]$, $C[20, 155]$ and $C[31, 155]$;
- Ring-based infrastructure: $R[5, 31]$ and $R[31, 5]$;
- Tree-based infrastructure: $T[5, 2, 5]$ and $T[3, 5, 5]$.

Figure 3 shows that the cluster has the worst performance. One can easily notice that when the cluster reaches an equilibrium (~ 2000), ~ 800 time units are needed to deliver a message to all components. We also observe that the number of nodes in the cluster has a minimal impact on this measure because

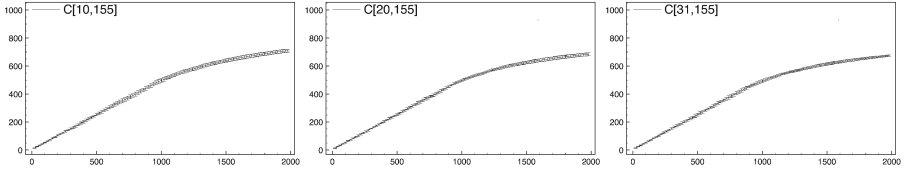


Fig. 3. CI scenario: Avg. Delivery Time for Cluster with 10, 20 and 31 servers.

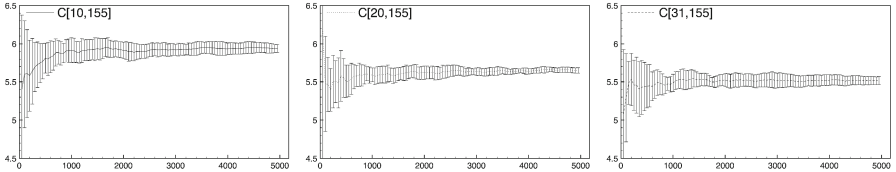


Fig. 4. CI scenario: Avg. Message Time Gap for Cluster with 10, 20 and 31 servers.

they all share the same input queue. The *Average Message Time Gap*, in Fig. 4, indicates that in the long run a component receives a message every $6/5.5$ time units.

Better performance can be obtained if the ring infrastructure is used. In the first two plots of Fig. 5 we report the average delivery time for the configurations $R[5, 31]$ and $R[31, 5]$. The last plot compares the average message time gap of the two configurations. In the first one, a message is delivered to all the components in 350 time units while in the second one 250 time units are needed. This indicates that increasing the number of nodes in the ring enhances performance. This is because in the ring all nodes cooperate to deliver a given message. Also the time gap decreases, i.e., a message is received every 2.6 and 1.8 time units.

Figure 6 shows how the *average delivery time* changes during the simulation for $T[5, 2, 5]$ and $T[3, 5, 5]$. The two configurations have exactly the same number of nodes (31) with a different arrangement. The two configurations work almost in the same way: a message is delivered to all the components in about 120 time units. Clearly, the tree is 5-time faster than the cluster and 2-time faster than the ring. Moreover, in the tree-based approach, a message is delivered to components

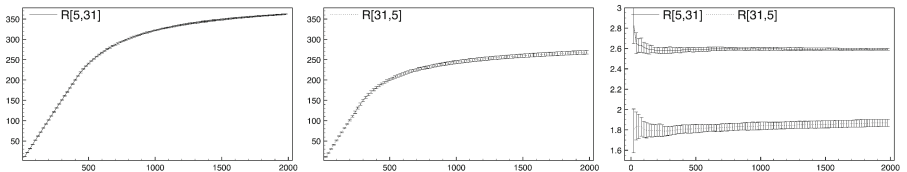


Fig. 5. CI scenario: Avg. Delivery Time and Avg. Message Time Gap for Ring with 5 and 31 servers.

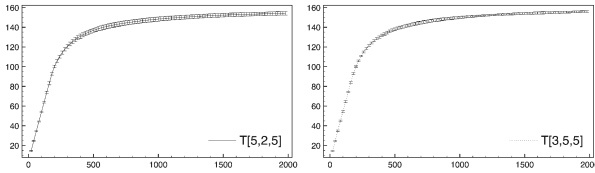


Fig. 6. CI scenario: Avg. Delivery Time: Tree/ $T[5, 2, 5]$ and $T[3, 5, 5]$.

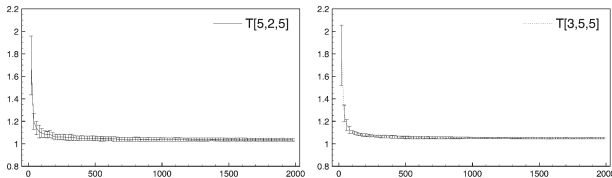


Fig. 7. CI scenario: Avg. Message Time Gap: Tree/ $T[5, 2, 5]$ and $T[3, 5, 5]$.

every ~ 1.1 time units as reported in Fig. 7. This means that messages in the tree are constantly delivered after an initial delay.

The results show that tree infrastructures offer the best performance; cluster-based ones do not work well while ring-based ones are in between the two.

5 Concluding Remarks, Future Work, and Related Work

The contribution of our paper is twofold: (i) the definition of a distributed tree-based infrastructure for coordinating attribute-based interaction and its actual implementation in Google Go; (ii) the proof of correctness of the proposed infrastructure and its performance evaluation. The results showed that the tree infrastructure has a better performance when compared with others in terms of minimising the average delivery time and maximising throughput.

As for future work, we plan to integrate (possibly) existing techniques for resilience to deal with imperfect communication links, dropped messages, and node’s failures. Apart from simulation, we will consider large and realistic case studies to investigate the actual performance of the current Go implementation.

We would like to conclude by relating to existing approaches. For implementations of attribute-based interaction, we refer to the Java-based [2] and the Erlang-based [10] implementations. As we mentioned before, these implementations are centralised while we are aiming for a distributed one.

Many approaches have been proposed to deal with distributed coordination, but they are difficult to compare as they differ in their assumptions, properties, objectives, or target applications [11]. However, they can be classified according to their ordering mechanisms. Below we relate to well-known approaches.

In the fixed sequencer approach [9], a single sequencer maintains the order of message delivery and components communicate by interacting only with the sequencer. The cluster infrastructure is a natural *distributed* extension; and also

the tree is a generalisation of this approach where instead of a single sequencer, we consider a propagation tree. The ordering decisions are resolved along tree paths. Actually, a propagation tree with depth 1 is a fixed sequencer.

The moving sequencer approach [7] avoids the bottleneck of a single sequencer by transferring the sequencer role among several nodes. Nodes form a logical ring and circulate a token, carrying a counter and a list of sequenced messages. Once token is received, a sequencer sequences its messages, sends all sequenced messages to its connected components, updates the token, and passes it along with sequenced messages to next node; thus the load is distributed among several nodes. However, the liveness of the algorithm depends on the token and, if the number of senders in one node is larger than others, fairness is hard to achieve. The ring-based infrastructure can be viewed as a generalisation of this technique where fairness is “resolved” by sharing a common counter.

In the privilege-based approach [8], senders circulate a token and each sender has to wait for the token. Upon receipt of token, the sender sequences its messages, sends them to destinations, and passes the token to the next sender. This approach is not suitable for open systems, since it assumes that all senders know each other. Also fairness is hard to achieve, e.g., some components send larger number of messages than others.

References

1. Abd Alrahman, Y., De Nicola, R., Loret, M.: On the power of attribute-based communication. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 1–18. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39570-8_1
2. Abd Alrahman, Y., De Nicola, R., Loret, M.: Programming of CAS systems by relying on attribute-based communication. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I. LNCS, vol. 9952, pp. 539–553. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_38
3. Abd Alrahman, Y., De Nicola, R., Loret, M.: Programming the Interactions of Collective Adaptive Systems by Relying on Attribute-based Communication. ArXiv e-prints, October 2017. <http://arxiv.org/abs/1711.06092>
4. Abd Alrahman, Y., De Nicola, R., Loret, M., Tiezzi, F., Vigo, R.: A calculus for attribute-based communication. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC 2015, pp. 1840–1845. ACM (2015). <https://doi.org/10.1145/2695664.2695668>
5. Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge (1986)
6. Albert, E., Lanese, I. (eds.): FORTE 2016. LNCS, vol. 9688. Springer, Cham (2016). <https://doi.org/10.1007/978-3-319-39570-8>
7. Chang, J.M., Maxemchuk, N.F.: Reliable broadcast protocols. *ACM Trans. Comput. Syst.* **2**, 251–273 (1984). <https://doi.org/10.1145/989.357400>
8. Cristian, F.: Asynchronous atomic broadcast. *IBM Tech. Discl. Bull.* **33**(9), 115–116 (1991)
9. Cristian, F., Mishra, S.: The pinwheel asynchronous atomic broadcast protocols. In: Second International Symposium on Autonomous Decentralized Systems, Proceedings, ISADS 1995, pp. 215–221. IEEE (1995). <https://doi.org/10.1109/ISADS.1995.398975>

10. De Nicola, R., Duong, T., Inverso, O., Trubiani, C.: AErlang at work. In: Steffen, B., Baier, C., van den Brand, M., Eder, J., Hinchey, M., Margaria, T. (eds.) SOFSEM 2017. LNCS, vol. 10139, pp. 485–497. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-51963-0_38
11. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: taxonomy and survey. *ACM Comput. Surv.* **36**, 372–421 (2004). <https://doi.org/10.1145/1041680.1041682>
12. Ferscha, A.: Collective adaptive systems. In: Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers, UbiComp/ISWC 2015 Adjunct, pp. 893–895. ACM, New York (2015). <https://doi.org/10.1145/2800835.2809508>
13. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. *J. ACM* **32**(2), 374–382 (1985). <https://doi.org/10.1145/3149.214121>
14. Jensen, T.R., Toft, B.: *Graph Coloring Problems*, vol. 39. Wiley, New York (1995)
15. Lopes, L., Silva, F., Vasconcelos, V.T.: A virtual machine for a process calculus. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702, pp. 244–260. Springer, Heidelberg (1999). https://doi.org/10.1007/10704567_15
16. Prasad, K.V.S.: A calculus of broadcasting systems. In: Abramsky, S., Maibaum, T.S.E. (eds.) CAAP 1991. LNCS, vol. 493, pp. 338–358. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-53982-4_19
17. Robertson, J.B.: Continuous-time Markov chains (W. J. Anderson). *SIAM Rev.* **36**(2), 316–317 (1994)
18. Sangiorgi, D., Walker, D.: *The PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, Cambridge (2003)
19. Schulze, T.P.: Efficient kinetic Monte Carlo simulation. *J. Comput. Phys.* **227**(4), 2455–2462 (2008). <http://www.sciencedirect.com/science/article/pii/S0021999107004755>
20. Vukolić, M.: The quest for scalable blockchain fabric: Proof-of-Work vs. BFT replication. In: Camenisch, J., Kesdoğan, D. (eds.) iNetSec 2015. LNCS, vol. 9591, pp. 112–125. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39028-4_9



Applied Choreographies

Saverio Giallorenzo¹(✉), Fabrizio Montesi¹, and Maurizio Gabbriellini²

¹ University of Southern Denmark, Odense, Denmark
saverio.giallorenzo@gmail.com

² Università di Bologna/Inria, Bologna, Italy

Abstract. Choreographic Programming is a paradigm for distributed programming, where high-level “Alice and Bob” descriptions of communications (choreographies) are used to synthesise correct-by-construction programs. However, implementations of choreographic models use message routing technologies distant from their related theoretical models (e.g., CCS/ π channels). This drives implementers to mediate discrepancies with the theory through undocumented, unproven adaptations, weakening the reliability of their implementations.

As a solution, we propose the framework of Applied Choreographies (AC). In AC, programmers write choreographies in a language that follows the standard syntax and semantics of previous works. Then, choreographies are compiled to a real-world execution model for Service-Oriented Computing (SOC). To manage the complexity of this task, our compilation happens in three steps, respectively dealing with: implementing name-based communications using the concrete mechanism found in SOC, projecting a choreography to a set of processes, and translating processes to a distributed implementation in terms of services.

1 Introduction

Background. In Choreographic Programming, programs are choreographies of communications used to synthesise correct implementations through an EndPoint Projection (EPP) procedure [1]. The generated code is guaranteed to follow the behaviour specified in the choreography and to be deadlock-free [2]. For these reasons, the communities of business processes and Service-Oriented Computing (SOC) widely adopted choreographies, using them in standards (e.g., WS-CDL [3] and BPMN [4]), languages [5–12], and type systems and logics [13–16].

Example 1. Below, we give a representative example of a choreography. In the example, we implement a simple business protocol among a client process c , a seller service located at l_s and a bank service located at l_b (locations are abstractions of network addresses, or URIs). At line 1, the client c asks the seller and the bank services to create two new processes, respectively s and b . The three processes c , s , and b can now communicate over a multiparty session k , intended as in Multiparty Session Types [13]: when a session is created, each process gets ownership of a statically-defined *role*, which identifies a message queue that

The original version of this chapter was revised: An acknowledgment has been added. The correction to this chapter is available at https://doi.org/10.1007/978-3-319-92612-4_11

the process uses to asynchronously receive messages from other processes. For simplicity, at line 1, we assign role **C** to process **c**, **S** to **s**, and **B** to **b**. At line 2, over session **k**, the client **c** invokes operation **buy** of the seller **s** with the name of a **product** it wishes to buy, which the seller stores in its local variable **x**. As usual, processes have local state and run concurrently. At line 3, the seller uses its internal function **mkOrder** to prepare an order (e.g., compute the price of the product) and sends it to the bank on operation **openTx**, for opening a payment transaction. At line 4, the client sends its credit card information **cc** to the bank on operation **pay**. Then, at line 5, the bank makes an internal choice on whether the payment can be performed (with internal function **closeTx**, which takes the local variables **cc** and **order** as parameters). The bank then notifies the client and the seller of the final outcome, by invoking them both either on operation **ok** or **ko**.

```

1  start k : c[C] <=> ls.s[S], lb.b[B];
2  k:c[C].product → s[S].buy(x);
3  k:s[S].mkOrder(x) → b[B].openTx(order);
4  k:c[C].cc → b[B].pay(cc);
5  if b.closeTx(cc, order)
6  { k:b[B] → c[C].ok(); k:b[B] → q[S].ok() }
7  else
8  { k:b[B] → c[C].ko(); k:b[B] → q[S].ko() }

```

Motivation. In previous definitions of EPP, both the choreography language and the target language abstract from how real-world frameworks support communications [2, 5, 14, 15, 17], by modelling communications as synchronisations on *names* (cf. [18, 19]). Thus, the implementations of choreographic programming [11, 12] significantly depart from their respective formalisations [2, 8]. In particular, the implemented EPPs realise channel creation and message routing with additional data structures and message exchanges [1, 20]. The specific communication mechanism used in these implementations is message correlation. Correlation is the reference message routing technology in Service-Oriented Computing (SOC)—the major field of application of choreographies—and it is supported by mainstream technologies (e.g., WS-BPEL [21], Java/JMS, C#/.NET). The gap between formalisations and implementations of choreographic programming can compromise its correctness-by-construction guarantee.

Contributions. We reduce the gap between choreographies and their implementations by developing Applied Choreographies, a choreographic framework consisting of three calculi: the Frontend Calculus (FC), which offers the well-known simplicity of abstract channel semantics to programmers; the Backend Calculus (BC), which formalises how abstract channels can be implemented on top of message correlation; and the Dynamic Correlation Calculus (DCC), an abstract model of Service-Oriented Computing where distributed services communicate through correlation. Differently from BC, DCC has no global view on the state of the system (which is instead distributed), and there are no multi-party synchronisation primitives.

Our main contribution is the definition of a behaviour-preserving compiler from choreographies in FC to distributed services in DCC, which uses BC as intermediate representation. This is the first correctness result of an end-to-end translation from choreographies to an abstract model based on a real-world communication mechanism. Our compiler proceeds in three steps:

- it projects (EPP) a choreography, describing the behaviours of many participants, into a composition of *modules* called *endpoint choreographies*, each describing the behaviour of a single participant;
- it generates the data structures needed by BC to support the execution of the obtained endpoint choreographies using message correlation;
- it synthesises a correct distributed implementation in DCC, where the multi-party synchronisations in choreographies are translated to correct distributed handshakes (consisting of many communications) on correlation data.

Full definitions and proof sketches are in [22].

2 Frontend Calculus

We start by presenting the Frontend Calculus (FC), which follows the structure of Compositional Choreographies [23]. The main novelty is in the semantics, which is based on a notion of deployment and deployment effects that will ease the definition of our translation. Remarkably, it also yields a new concise formalisation of asynchrony in choreographies.

Syntax. In the syntax of FC (Fig. 1), C denotes an FC program. FC programs are choreographies, as in Example 1, and we often refer to them as Frontend choreographies. A choreography describes the behaviour of some processes. Processes, denoted $p, q \in \mathcal{P}$, are intended as usual: they are independent execution units running concurrently and equipped with local variables, denoted $x \in \text{Var}$. Message exchanges happen through a session, denoted $k \in \mathcal{K}$, which acts as a communication channel. Intuitively, a session is an instantiation of a protocol, where each process is responsible for implementing the actions of a role defined in the protocol. We denote roles with $A, B \in \mathcal{R}$. A process can create new processes and sessions at runtime by invoking service processes (services for short). Services are always available at fixed locations, denoted $l \in \mathcal{L}$, meaning that they can be used multiple times (in process calculus terms, they act as replicated processes [24]). Locations are novel to choreographies: they are addresses of always-available services able to create processes at runtime (cf. public channels in [2, 23]).

Processes communicate by exchanging messages. A message consists of two elements: (i) a payload, representing the data exchanged between two processes; and (ii) an operation, which is a label used by the receiver to determine what it should do with the message—in object-oriented programming, these labels are called method names [25]; in service-oriented computing, labels are typically called operations as in here. Operations are denoted $o \in \mathcal{O}$.

In the rest of the section, we comment the syntax of FC and present its semantics. However, before doing that, we dedicate the next paragraph to illustrate with an example the purpose and relationship between complete and partial actions.

$C ::= \eta; C$	(<i>seq</i>)		$C_1 \mid C_2$	(<i>par</i>)
$\text{if } p.e \{C_1\} \text{ else } \{C_2\}$	(<i>cond</i>)		$\mathbf{0}$	(<i>inact</i>)
$\text{def } X = C' \text{ in } C$	(<i>rec</i>)		X	(<i>call</i>)
$k:A \longrightarrow q[B].\{o_i(x_i); C_i\}_{i \in I}$	(<i>recv</i>)			
$\eta ::= k:p[A].e \longrightarrow q[B].o(x)$	(<i>com</i>)		$\text{start } k : p[A] \langle \langle \rangle \rangle \overline{\mathbf{l}.q[B]}$	(<i>start</i>)
$k:p[A].e \longrightarrow B.o$	(<i>send</i>)		$\text{req } k : p[A] \langle \langle \rangle \rangle \overline{\mathbf{l}.B}$	(<i>req</i>)
			$\text{acc } k : \overline{\mathbf{l}.q[B]}$	(<i>acc</i>)

Fig. 1. Frontend Calculus, syntax.

Complete and Partial Actions. As in [23], FC supports modular development by allowing choreographies, say C and C' , to be composed in parallel, written $C \mid C'$. A parallel composition of choreographies is also a choreography, which can thus be used in further parallel compositions. Composing two choreographies in parallel allows the processes in the two choreographies to interact over shared location and session names. In particular, we distinguish between two kinds of terms inside of a choreography: complete and partial actions. A complete action is internal to the system defined by the choreography, and thus does not have any external dependency. By contrast, a partial action defines the behaviour of some processes that need to interact with another choreography in order to be executed. Therefore, a choreography containing partial actions needs to be composed with other choreographies that provide compatible partial actions. To exemplify the distinction between complete and partial actions, let us consider the case of a single communication between two processes.

<i>Complete interaction</i>	<i>Composed partial actions</i>
$k:c[C].\text{product} \longrightarrow s[S].\text{buy}(x)$	$k:c[C].\text{product} \longrightarrow S.\text{buy}$ $k:C \longrightarrow s[S].\text{buy}(x)$

Above, on the left we have the communication statement as written at line 2 of Example 1. This is a complete action: it defines exactly all the processes that should interact (c and s). On the right, we implement the same action as the parallel composition of two *modules*, i.e., choreographies with partial actions. At the left of the parallel we write a send action, performed by process c to role S over session k , at the right of the parallel we write the complementary reception from a role C and performed by process s over the session k . More specifically, we read the send action as “process c sends a message as role C with payload product for operation buy to the process playing role S on session k ”. Dually, we

read the receive action as “process s receives a message for role S and operation buy over session k and stores the payload in variable x ”. The compatible roles, session, and operation used in the two partial actions make them compliant. Thus, the choreography on the left is operationally equivalent to the one on the right. Observe that partial actions do not mention the name of the process on the other end—for example, the send action performed by process c does not specify that it wishes to communicate with process s precisely. This supports some information hiding: a partial action in a choreography can interact with partial actions in other choreographies, independently from the process names used in the latter. Expressions and variables used by senders and receivers are also kept local to statements that define local actions.

By equipping FC with both partial and complete actions, we purposefully made FC non-minimal. However, while a minimal version of FC (i.e., equipped with either partial or complete actions) can capture well-known choreographic models like [2, 5–8], we included both kinds of actions to describe a comprehensive language for programmers. On the one hand, complete actions offer a concise syntax to express closed systems, as found in other choreographic models. On the other hand, partial actions support compositionality in Frontend choreographies, allowing developers to write FC modules separately and possibly reuse the same module in multiple compositions.

Complete Actions. We start commenting the syntax of FC from complete actions, marked with a `shade` in Fig. 1. In term (start) , process p starts a new session k together with some new processes \tilde{q} (\tilde{q} is assumed non-empty). Process p , called *active process*, is already running, whereas each process q in $\tilde{l}.q$, called *service process*, is dynamically created at the respective location l . Each process is annotated with the role it plays in the session. Term (com) models a communication: on session k , process p sends to process q a message for its operation o ; the message carries the evaluation of expression e (we assume expressions to consist of standard computations on local variables) on the local state of p whilst x is the variable where q will store the content of the message.

Partial Actions. Partial actions correspond to the terms obtained by respectively splitting the (start) and (com) complete terms into their partial counterparts. In term (req) , process p requests some external services respectively located at \tilde{l} , to create some external processes and start a new session k . By “external”, we mean that the behaviour of such processes is defined in a separate choreography module, to be composed in parallel. Role annotations follow those of term (start) . Term (acc) is the dual of (req) and defines a choreography module that provides the implementation of some service processes. We assume (acc) terms to always be at the top level (not guarded by other actions), capturing always-available choreography modules. In term (send) , process p sends a message to an external process that plays B in session k . Dually, in term (recv) , process q receives a message for one of the operations o_i from an external process playing role A in session k , and then proceeds with the corresponding continuation (cf. [26]).

Other Terms. In a conditional (*cond*), process p evaluates a condition e in its local state to choose between the continuations C_1 and C_2 . Term (*par*) is standard parallel composition, which allows partial actions in two choreographies C_1 and C_2 to interact. Respectively, terms (*def*), (*call*), and (*inact*) model the definition of recursive procedures, procedure calls, and inaction. Some terms bind identifiers in continuations. In terms (*start*) and (*acc*), the session identifier k and the process identifiers \bar{q} are bound (as they are freshly created). In terms (*com*) and (*recv*), the variables used by the receiver to store the message are bound (x and all the x_i , respectively). In term (*req*), the session identifier k is bound. Finally, in term (*def*), the procedure identifier X is bound. In the remainder, we omit $\mathbf{0}$ or irrelevant variables (e.g., in communications with empty messages). Terms (*com*), (*send*), and (*recv*) include role annotations only for clarity reasons; roles in such terms can be inferred, as shown in [1].

Semantics. The semantics of FC follows the standard principles of asynchronous multiparty sessions (cf. [13]) and it is formalised in terms of reductions of the form $D, C \rightarrow D', C'$, where D is a deployment. Deployments store the states of processes and the message queues that support message exchange in sessions. We start by formalising the notion of deployment.

Deployment. As in multiparty session types [13], we equip each pair of roles in a session with a dedicated asynchronous queue to communicate in each direction. Formally, let $\mathcal{Q} = \mathcal{K} \times \mathcal{R} \times \mathcal{R}$ be the set of all *queue identifiers*; we write $k[A]B \in \mathcal{Q}$ to identify the queue from A to B in session k . Now, we define D as an overloaded partial function defined by cases as the sum of two partial functions $f_s : \mathcal{P} \rightarrow \text{Var} \rightarrow \text{Val}$ and $f_q : \mathcal{Q} \rightarrow \text{Seq}(\mathcal{O} \times \text{Val})$ whose domains and co-domains are disjoint: $D = f_s(z)$ if $z \in \mathcal{P}$, $f_q(z)$ if $z \in \mathcal{Q}$.

Function f_s maps a process p to its state, which is a partial function from variables $x, y \in \text{Var}$ to values $v \in \text{Val}$. Function f_q stores the queues used in sessions. Each queue is a sequence of messages $\tilde{m} = m_1 :: \dots :: m_n \mid \varepsilon$ (ε is the empty queue), where each message $m = (o, v) \in \mathcal{O} \times \text{Val}$ contains the operation o it was received on and the payload v .

Programmers do not deal with deployments. We assume that choreographies without free session names start execution with a *default deployment* that contains empty process states. Let $\mathbf{fp}(C)$ return the set of free process names in C and, with abuse of notation, \emptyset be the undefined function for any given signature.

Definition 1 (Default Deployment). *Let C be a choreography without free session names. The default deployment of D for C is defined as the function mapping all free names in C to empty states, i.e., $D[p \mapsto \emptyset \mid p \in \mathbf{fp}(C)]$.*

Reductions. In our semantics, choreographic actions have effects on the state of a system—deployments change during execution. At the same time, a deployment also determines which choreographic actions can be performed. For example, a communication from role A to role B over session k requires a queue $k[A]B$ to exist in the deployment of the system.

We formalise the notion of which choreographic actions are allowed by a deployment and their effects using transitions of the form $D \xrightarrow{\delta} D'$, read “the deployment D allows for the execution of δ and becomes D' as the result”. Actions δ are defined by the following grammar: **start** $k: p[A] \Leftrightarrow \overline{l.q[B]}$, session start, $k: p[A].e \rightarrow B.o$, send in session, $k:A \rightarrow q[B].o(x)$, receive in session.

We report the rules defining $D \xrightarrow{\delta} D'$ in the top-half of Fig. 2.

$$\begin{array}{l}
\llbracket P_{\text{Start}} \rrbracket \quad D \xrightarrow{\text{start } k: p[A] \Leftrightarrow \overline{l.q[B]}} D [q \mapsto \emptyset \mid q \in \{\tilde{q}\}] [k[C]E \mapsto \varepsilon \mid \{C, E\} \subseteq \{A, \tilde{B}\}] \\
\llbracket P_{\text{Send}} \rrbracket \quad e \downarrow_{D(p)} v \Rightarrow D [k[A]B \mapsto m] \xrightarrow{k p[A].e \rightarrow B.o} D [k[A]B \mapsto \tilde{m} :: (o, v)] \\
\llbracket P_{\text{Recv}} \rrbracket \quad D [k[A]B \mapsto (o, v) :: \tilde{m}] \xrightarrow{k A \rightarrow q[B].o(x)} D [k[A]B \mapsto \tilde{m}, q \mapsto D(q)[x \mapsto v]] \\
\hline
\llbracket C_{\text{Start}} \rrbracket \quad \left\{ \begin{array}{l} \delta = \text{start } k: p[A] \Leftrightarrow \overline{l.q[B]} \\ \wedge D \# k', \tilde{r} \wedge D \xrightarrow{\delta [k'/k][\tilde{r}/\tilde{q}]} D' \end{array} \right. \Rightarrow D, \delta; C \rightarrow D', C[k'/k][\tilde{r}/\tilde{q}] \\
\llbracket C_{\text{Send}} \rrbracket \quad \eta = k: p[A].e \rightarrow B.o \wedge D \xrightarrow{\eta} D' \Rightarrow D, \eta; C \rightarrow D', C \\
\llbracket C_{\text{Recv}} \rrbracket \quad \left\{ \begin{array}{l} \delta = k: A \rightarrow q[B].o_j(x_j) \\ \wedge j \in I \wedge D \xrightarrow{\delta} D' \end{array} \right. \Rightarrow D, k: A \rightarrow q[B].\{o_i(x_i); C_i\}_{i \in I} \rightarrow D', C_j \\
\llbracket C_{\text{Eq}} \rrbracket \quad \left\{ \begin{array}{l} \mathcal{R} \in \{\equiv_c, \simeq_c\} \wedge C_1 \mathcal{R} C'_1 \\ \wedge D, C'_1 \rightarrow D', C'_2 \wedge C'_2 \mathcal{R} C_2 \end{array} \right. \Rightarrow D, C_1 \rightarrow D', C_2
\end{array}$$

Fig. 2. FC: top-half deployment transitions, bottom-half semantics (selected).

Rule $\llbracket P_{\text{Start}} \rrbracket$ states that the creation of a new session k between an existing process p and new processes \tilde{q} results in updating the deployment with: a new (empty) state for each of the new processes q in \tilde{q} ($[q \mapsto \emptyset \mid q \in \{\tilde{q}\}]$); and a new (empty) queue between each pair of distinct roles in the session ($[k[C]E \mapsto \varepsilon \mid \{C, E\} \subseteq \{A, \tilde{B}\}]$). Rule $\llbracket P_{\text{Send}} \rrbracket$ models the effect of a send action. In the premise, we use the auxiliary function \downarrow to evaluate the local expression e in the state of process p , obtaining the value v to use as message payload. Then, in the conclusion, we append a message with its payload— (o, v) —to the end of the queue from the sender’s role to the receiver’s role ($k[A]B$). We assume that function \downarrow always terminates—in practice, this can be obtained by using timeouts. Rule $\llbracket P_{\text{Recv}} \rrbracket$ models the effect of a reception. If the queue $k[A]B$ contains in its head a message on operation o , it is always possible to remove it from the queue and to store its value v under variable x in the state of the receiver.

Using deployment transitions, we can now define the rules for reductions $D, C \rightarrow D', C'$. We call a configuration D, C a *running choreography*. The reduction relation \rightarrow for FC is the smallest relation closed under the (excerpt of) rules given in the bottom-half of Fig. 2. Rule $\llbracket C_{\text{Start}} \rrbracket$ creates a new session, by ensuring

that both the new session name k' and new processes \tilde{r} are fresh wrt D ($D\#k', \tilde{r}$). We use the fresh names in the continuation C , by using a standard substitution $C[k'/k][\tilde{r}/\tilde{q}]$. Rule $[\text{Send}]$ reduces a send action, if it is allowed by the deployment. Rule $[\text{Recv}]$ reduces a message reception, if the deployment allows for receiving a message on one of the branches in the receive term ($j \in I$). Recalling the corresponding rule $[\text{Recv}]$, this can happen only if the deployment D has a message for operation o_j in the queue $k[A]B$. Rule $[\text{Eq}]$ closes \rightarrow under the congruences \equiv_c and \simeq_c . Structural congruence \equiv_c is the smallest congruence supporting α -conversion, recursion unfolding, and commutativity and associativity of parallel composition. The swap relation \simeq_c is the smallest congruence able to exchange the order of non-interfering concurrent actions. Rule $[\text{Eq}]$ also enables the reduction of complete communications on (*com*) terms with the equivalence

$$k:p[A].e \longrightarrow q[B].o(x); C \equiv_c k:p[A].e \longrightarrow B.o; k:A \longrightarrow q[B].\{o(x); C\}$$

unfolding complete communications into the corresponding send and receive terms.

Typing. We equip FC with a typing discipline based on multiparty session types [13, 26], which checks that partial actions composed in parallel are compatible. Our typing discipline is a straightforward adaptation of that presented for Compositional Choreographies [23], so we omit most details here (reported in [22]). However, the type system also gives us important information that will be critical in the compilation that we will develop later. Specifically, we mainly use types to keep track of the location that each process has been created at, the types of variables, the roles played by processes in open sessions, the role played by the processes spawned at each location in a choreography, and the status of message queues. Keeping track of this information is straightforward; hence, for brevity, we simply report the definition of the main elements of the typing environment Γ that we use to store it.

$$\Gamma ::= \Gamma, \tilde{l} : G\langle A|\tilde{B}|\tilde{C}\rangle \mid \Gamma, k[A] : T \mid \Gamma, p : k[A] \mid \Gamma, p.x : U \mid \Gamma, p@l \mid k[B]A : T \mid \emptyset$$

In Γ , a service typing $\tilde{l} : G\langle A|\tilde{B}|\tilde{C}\rangle$ types with the (standard, from [23, 26]) global type G any session started by contacting the services at the locations \tilde{l} . Role A is the role of the active process, whereas roles \tilde{B} are the respective roles of the service processes located at \tilde{l} . (The roles \tilde{C} are used to keep track of whether the implementation of some roles is provided by external choreography modules.) A session typing $k[A] : T$ defines that role A in session k follows the local type T . An ownership typing $p : k[A]$ states that process p owns the role A in session k . A location typing $p@l$ states that process p runs at location l . The typing $p.x : U$ states that variable x has type U in the state of process p . Finally, the buffer typing $k[A]B : T$ states that the queue $k[A]B$ contains a sequence of messages typed by the local type T .

A typing judgement $\Gamma \vdash D, C$ establishes that D and C are typed according to Γ . If such a Γ exists, we say that D, C is well-typed.

Thanks to Γ , we can define a formal notion of coherence \mathbf{co} , useful to check if a given set of Frontend modules can correctly execute a typed session:

Definition 2 (Coherence). $\mathbf{co}(\Gamma)$ holds iff $\forall k \in \Gamma, \exists G$ s.t.

- $\tilde{\Gamma} : G \langle A | \tilde{B} | \tilde{C} \rangle \in \Gamma \wedge \tilde{C} = \tilde{B}$ and
- $\forall A \in \mathbf{roles}(G), k[A] : T \in \Gamma \wedge T = \llbracket G \rrbracket_A$
 $\wedge \forall B \in \mathbf{roles}(G) \setminus \{A\}, \Gamma \vdash k[A]B : \llbracket G \rrbracket_B^A$

Coherence checks that (i) all services needed to start new sessions are present and (ii) all the roles in every open session are correctly implemented by some processes. To do this, given a global type G , \mathbf{co} uses function $\llbracket G \rrbracket_A$ to extract the local type of a role A in G and function $\llbracket G \rrbracket_B^A$ to extract the local type of the queue where role B receives from role A in G .

Well-typed Frontend choreographies that contain all necessary modules never deadlock.

Theorem 1 (Deadlock freedom). $\Gamma \vdash D, C$ and $\mathbf{co}(\Gamma)$ imply that either (i) $C \equiv \mathbf{0}$ or (ii) there exist D' and C' such that $D, C \rightarrow D', C'$.

3 Backend Calculus

We now present the *Backend Calculus* (BC). Formally, the syntax of programs in BC is the same as that of FC. The only difference between BC and FC is in the semantics: we replace the notions of deployment and deployment effects with new versions that formalise message exchanges based on message correlation, as found in Service-Oriented Computing (SOC) [21].

Deployments in BC. Deployments in BC capture communications in SOC, where data trees are used to correlate messages to input queues. We first informally introduce correlation.

Message Correlation. Processes in SOC run within services and communicate asynchronously: each process can retrieve messages from an unbound number of FIFO queues, managed by its enclosing service. To identify queues, services use some data, called *correlation key*. When a service receives a message from the network, it inspects its content looking for a correlation key that points one of its queues. If a queue can be found, the message is enqueued in its tail. As noted in the example, messages in SOC contain correlation keys as either part their payload or in some separate header. As in [27], we abstract from such details.

Data and Process state. Data in SOC is structured following a tree-like format, e.g., XML or JSON. We use trees to represent both the payload of messages and the state of running processes (as in, e.g., BPEL [21] and Jolie [28]). Formally, we consider rooted trees $t \in \mathcal{T}$, where edges are labelled by names, ranged over by \underline{x} , and \emptyset is the empty tree. We assume that all outgoing edges of a node have distinct labels and that only leaves contain values of type $Val \cup \mathcal{L}$, i.e., basic data (**int**, **str**, ...) or locations. Variables x are paths to traverse a tree:

$x, y ::= \underline{x}.x \mid \varepsilon$, where ε is the empty path (often omitted). Given a path x and a tree t , $x(t)$ is the node reached by following x in t ; otherwise, $x(t)$ is undefined. Abusing notation, when $x(t)$ is a leaf, then $x(t)$ denotes also the value of the node. To manipulate trees, we will use the (total) replacement operator $t \triangleleft (x, t')$. If $x(t)$ is defined, $t \triangleleft (x, t')$ returns the tree obtained by replacing in t the subtree rooted in $x(t)$ by t' . If $x(t)$ is undefined, $t \triangleleft (x, t')$ adds the smallest chain of empty nodes to t such that $x(t)$ is defined and it stores t' in $x(t)$.

Backend Deployment. We can now define the notion of deployment for BC, denoted \mathcal{D} . Formally, \mathcal{D} is an overloaded partial function defined by cases as the sum of three partial functions $g_l : \mathcal{L} \rightarrow \text{Set}(\mathcal{P})$, $g_m : \mathcal{L} \times \mathcal{T} \rightarrow \text{Seq}(\mathcal{O} \times \mathcal{T})$, and $g_s : \mathcal{P} \rightarrow \mathcal{T}$ whose domains and co-domains are disjoint: $\mathcal{D}(z) = g_l(z)$ if $z \in \mathcal{L}$, $g_m(z)$ if $z \in \mathcal{L} \times \mathcal{T}$, and $g_s(z)$ if $z \in \mathcal{P}$.

Function g_l maps a location to the set of processes running in the service at that location. Given l , we read $\mathcal{D}(l) = \{p_1, \dots, p_n\}$ as “the processes p_1, \dots, p_n are running at the location l ” (we assume processes to run at most at one location). Function g_m maps a couple location-tree to a message queue. This reflects message correlation as informally described above, where a queue resides in a service, i.e., at its location, and is pointed by a correlation key. Given a couple $l : t$, we read $\mathcal{D}(l : t) = \tilde{m}$ as “the queue \tilde{m} resides in a service at location l and is pointed by correlation key t ”. The queue \tilde{m} is a sequence of messages $\tilde{m} ::= m_1 :: \dots :: m_n \mid \varepsilon$ and a message of the queue is $m ::= (o, t)$, where t is the payload of the message and o is the operation on which the message was received. Function g_s maps a process to its local state. Given a process p , the notation $\mathcal{D}(p) = t$ means that p has local state t .

Deployment Effects in BC. In BC, we replace FC deployment effects (i.e., the rules for $\mathcal{D} \xrightarrow{\delta} \mathcal{D}'$) with the ones reported in Fig. 3, commented below.

$$\begin{array}{c}
 \frac{p \in \mathcal{D}(l) \quad \mathcal{D} \xrightarrow{\text{sup}(l.p[A], \overline{l.q.[B]})} \mathcal{D}'}{\mathcal{D} \xrightarrow{\text{start } k: p[A] \leftrightarrow \overline{l.q.[B]}} \mathcal{D}'} \quad [P]_{\text{Start}} \\
 \\
 \frac{q_1 \in \mathcal{D} \quad \textcircled{1} \quad j \in I \setminus \{i\} \quad \underline{B}_i.l(t) = l_i \quad \textcircled{2} \quad \underline{B}_i.B_j(t) = t_{ij} \quad \textcircled{3} \quad l_j : t_{ij} \notin \mathcal{D} \quad \textcircled{4}}{\mathcal{D}_1 = \mathcal{D} [l_i \mapsto \mathcal{D}(l_i) \cup \{q_i\}] \quad \textcircled{5} \quad \mathcal{D}_2 = \mathcal{D}_1 [l_i : t_{ij} \mapsto \varepsilon] \quad \textcircled{6}} \quad [P]_{\text{Sup}} \\
 \mathcal{D} \xrightarrow{\text{sup}(\overline{l_i.q_i.[B_i]} \quad t_{i \in I})} \mathcal{D}_2 [q_1 \mapsto \mathcal{D}_2(q_1) \triangleleft (k, t)] \quad \textcircled{7} [q_h \mapsto \{k : t\} \mid h \in I \setminus \{q_1\}] \quad \textcircled{8} \\
 \\
 \frac{l = \underline{k.B.l}(\mathcal{D}(p)) \quad t_c = \underline{k.A.B}(\mathcal{D}(p)) \quad e \downarrow_{\mathcal{D}(p)} t_m}{\mathcal{D} \xrightarrow{k \ p[A].e \rightarrow B.o} \mathcal{D} [l : t_c \mapsto \mathcal{D}(l : t_c) :: (o, t_m)]} \quad [P]_{\text{Send}} \\
 \\
 \frac{t_c = \underline{k.A.B}(\mathcal{D}(q)) \quad q \in \mathcal{D}(l) \quad \mathcal{D}(l : t_c) = (o, t_m) :: \tilde{m} \quad \mathcal{D}' = \mathcal{D} [l : t_c \mapsto \tilde{m}]}{\mathcal{D} \xrightarrow{k \ \wedge \rightarrow q[B].o(x)} \mathcal{D}' [q \mapsto \mathcal{D}'(q) \triangleleft (\underline{x}, t_m)]} \quad [P]_{\text{Recv}}
 \end{array}$$

Fig. 3. Deployment effects for Backend Choreographies.

Rule $[\mathcal{P}|_{\text{start}}]$ simply retrieves the location of process p (the one that requested the creation of session k) and uses Rule $[\mathcal{P}|_{\text{sup}}]$ to obtain the new deployment \mathcal{D}' that supports interactions over session k . Namely, \mathcal{D}' is an updated version of \mathcal{D} with: (i) the newly created processes for session k and (ii) the queues used by the new processes and p to communicate over session k . In addition, in \mathcal{D}' , (iii) the new processes and p contain in their states a structure, rooted in \underline{k} and called *session descriptor*, that includes all the information (correlation keys and the locations of all involved processes) to support correlation-based communication in session k . Formally, this is done by Rule $[\mathcal{P}|_{\text{sup}}]$ where we ① retrieve the starter process, here called q_1 , which is the only process already present in \mathcal{D} . Then, given a tree t , we ensure it is a proper session descriptor for session k , i.e., that:

- ② t contains the location l_i of each process. The location is stored under path $\underline{B_i.l}$, where B_i is the role played by the i -th process in the session;
- ③ t contains a correlation key t_{ij} for each ordered couple of roles B_i, B_j under path $\underline{B_i.B_j}$, such that ④ there is no queue in \mathcal{D} at location l_j pointed by correlation key t_{ij} ;

Finally, we assemble the update of \mathcal{D} in four steps:

- ⑤ we obtain \mathcal{D}_1 by adding in \mathcal{D} the processes q_2, \dots, q_n at their respective locations;
- ⑥ we obtain \mathcal{D}_2 by adding to \mathcal{D}_1 an empty queue ε for each couple $l_j : t_{ij}$;
- ⑦ we update \mathcal{D}_2 to store in the state of (the starter) process q_1 the session support t under path \underline{k} ;
- ⑧ we further update \mathcal{D}_2 such that each new created process (q_2, \dots, q_n) has in its state just the session descriptor t rooted under path \underline{k} .

We deliberately define in $[\mathcal{P}|_{\text{sup}}]$ the session descriptor t with a set of constrains on data, rather than with a procedure to obtain the data for correlation. In this way, our model is general enough to capture different methodologies for creating correlation keys (e.g., UUIDs or API keys).

Rule $[\mathcal{P}|_{\text{send}}]$ models the sending of a message. From left to right of the premises: we retrieve the location l of the receiver B from the state of the sender p ; we retrieve the correlation key t_c in the state of p (playing role A) to send messages to role B ; we compute the payload of the message by evaluating the expression e against the local state of the sender p . Then we obtain the updated deployment by adding message (o, t_m) in the queue pointed by $l : t_c$ that we found via correlation.

Rule $[\mathcal{P}|_{\text{recv}}]$ models a reception. From left to right of the premises: we find the correlation key t_c for the queue that q (playing role B) uses to receive from A in session k ; we retrieve the location l of q ; we access the queue pointed by $l : t_c$ and retrieve message (o, t_m) ; we obtain a partial update of \mathcal{D} in \mathcal{D}' removing (o, t_m) from the interested queue; we obtain the updated deployment by storing the payload t_m in the state of q under path \underline{x} .

Encoding FC to BC. Runtime terms D, C in FC can be translated to BC simply by encoding D to an appropriate Backend deployment, since the syntax

of choreographies is the same. For this translation, we need to know the roles played by processes and their locations, which is not recorded in D . We extract this information from the typing of C .

Definition 3 (Encoding FC in BC). *Let D, C be well-typed: $\Gamma \vdash D, C$. Then, its Backend encoding is defined as $\langle\langle D \rangle\rangle^\Gamma, C$, where $\langle\langle D \rangle\rangle^\Gamma$ is given by the algorithm in Listing 1.1.*

```

1   $\langle\langle D \rangle\rangle^\Gamma = \mathcal{D} := \emptyset;$ 
2  foreach  $p@l$  in  $\Gamma. \mathcal{D} := \mathcal{D} [ l \mapsto \mathcal{D}(l) \cup \{p\} ]$ ;  $\mathcal{D} := \mathcal{D} [ p \mapsto \emptyset ]$ 
3  foreach  $p.x: U$  in  $\Gamma. \mathcal{D} := \mathcal{D} [ p \mapsto \mathcal{D}(p) \triangleleft (\underline{x}, \mathcal{D}(p)(x)) ]$ ;
4  foreach  $\{ p: k[A] \ q: k[B], q@l \}$  in  $\Gamma. t := \text{fresh}(\mathcal{D}, l)$ ;
5   $\mathcal{D} := \mathcal{D} [ l: t \mapsto \mathcal{D}(k[A]B) ]$ ;
6   $\mathcal{D} := \mathcal{D} [ p \mapsto \mathcal{D}(p) \triangleleft (\underline{k.A.B}, t) ]$ ;  $\mathcal{D} := \mathcal{D} [ p \mapsto \mathcal{D}(p) \triangleleft (\underline{k.B.l}, l) ]$ ;
7   $\mathcal{D} := \mathcal{D} [ q \mapsto \mathcal{D}(q) \triangleleft (\underline{k.A.B}, t) ]$ ;  $\mathcal{D} := \mathcal{D} [ q \mapsto \mathcal{D}(q) \triangleleft (\underline{k.B.l}, l) ]$ 
8  return  $\mathcal{D}$ 

```

Listing 1.1. Encoding Frontend to Backend Deployments.

Commenting the algorithm of $\langle\langle D \rangle\rangle^\Gamma$, at line 2 it includes in \mathcal{D} all (located) processes present in D (and typed in Γ) and instantiate their empty states. At line 3 it translates the state (i.e., the association *Variable-Value*) of each process in D to its correspondent tree-shaped state in \mathcal{D} . At lines 4–7, for each ongoing session in D (namely, for each couple of processes, each playing a role in a given session), it sets the proper correlation keys and queues in \mathcal{D} and, for each queue, it imports and translates the sequence of related messages.

The encoding from FC to BC guarantees a strong operational correspondence.

Theorem 2 (Operational Correspondence (FC \leftrightarrow BC)). *Let $\Gamma \vdash D, C$. Then:*

1. (Completeness) $D, C \rightarrow D', C'$ implies $\langle\langle D \rangle\rangle^\Gamma, C \rightarrow \langle\langle D' \rangle\rangle^{\Gamma'}, C'$ for some Γ' s.t. $\Gamma' \vdash D', C'$;
2. (Soundness) $\langle\langle D \rangle\rangle^\Gamma, C \rightarrow \mathcal{D}, C'$ implies $D, C \rightarrow D', C'$ and $\mathcal{D} = \langle\langle D' \rangle\rangle^{\Gamma'}$ for some Γ' s.t. $\Gamma' \vdash D', C'$.

4 Dynamic Correlation Calculus

We now introduce the Dynamic Correlation Calculus (DCC), the target language of our compilation. To define DCC, we considered a previous formal model for Service-Oriented Computing, based on correlation [27]. However, we found the calculus in [27] too simple for our purposes: there, each process has only one message queue, while here we need to manage many queues per process (as in our Backend deployments). Hence, to define DCC, we basically extend the calculus in [27] to let processes create and receive from multiple queues. Beside the requirement of this work, many languages for SOC (e.g., BPEL [21]) let processes create and receive from multiple queues, which makes DCC a useful reference calculus in general.

Syntax. The syntax of DCC, reported in Fig. 4, comprises two layers: *Services*, ranged over by S , and *Processes*, ranged over by P .

In the syntax of services, term (srv) is a service, located at l , with a *Start Behaviour* B_s and running processes P (both described later on) and a queue map M . The queue map is a partial function $M : \mathcal{T} \rightarrow Seq(\mathcal{O} \times \mathcal{T})$ that, similarly to function g_m in BC deployments, associates a correlation key t to a message queue. We model messages as in BC: a message is a couple (o, t) where o is the operation on which the message has been received and t the payload of the message. Services are composed in parallel in term (net) .

<i>Services</i>	$S ::=$	$\langle B_s, P, M \rangle_l$	(srv)	
		$S \mid S'$	(net)	
<i>Start Behaviour</i>	$B_s ::=$	$!(x); B$	$(acct)$	
		$\mathbf{0}$	$(inact)$	
<i>Processes</i>	$P ::=$	$B \cdot t$	$(prcs)$	
		$P \mid P'$	(par)	
<i>Behaviours</i>				
$B ::=$	$?@e_1(e_2); B$	$(reqst)$	$ o@e_1(e_2) \text{ to } e_3; B$	$(output)$
	$\sum_i [o_i(x_i) \text{ from } e] \{B_i\}$	$(choice)$	$\text{if } e \{B_1\} \text{ else } \{B_2\}$	$(cond)$
	$\text{def } X = B' \text{ in } B$	(def)	$\mathbf{0}$	$(inact)$
	$\nu x; B$	$(newque)$	X	$(call)$
	$x = e; B$	$(assign)$		

Fig. 4. Dynamic Correlation Calculus, syntax.

Concerning behaviours, in DCC we distinguish between start behaviours B_s and process behaviours B . Process behaviours define the general behaviour of processes in DCC, as described later on. Start behaviours use term $!(x)$ to indicate the availability of a service to generate new local processes on request. At runtime, the start behaviour B_s of a service is activated by the reception of a dedicated message that triggers the creation of a new process. The new process has (process) behaviour B , which is defined in B_s after the $!(x)$ term, and an empty state. The content of the request message is stored in the state of the newly created process, under the bound path x . As in BC, also in DCC paths are used to access process states.

Operations (o), procedures (X), variables (x , which are paths), and expressions (e , evaluated at runtime on the state of the enclosing process) are as in BC. Terms $(choice)$ and $(output)$ model communications. In a $(choice)$, when a message can be received from one of the operations o_i from the queue correlating with e , the process stores under x_i the received message, it discards all other inputs, and executes the continuation B_i . When only one input is available in a $(choice)$, we use the contracted form $o(x) \text{ from } e; B$. Term $(output)$ sends a message on operation o , with content e_2 , while e_1 defines the location of the service where the addressee (process) is running and e_3 is the key that correlates

with the receiving queue of the addressee. Term (*reqst*) is the dual of (*acpt*) and asks the service located at e_1 to spawn a new process, passing to it the message in e_2 . Term (*queue*) models the creation of a new queue that correlates with the key contained in variable x . Other terms are standard.

Semantics. In Fig. 5, we report a selection of the rules defining the semantics of DCC, a relation \rightarrow closed under a (standard) structural congruence \equiv that supports commutativity and associativity of parallel composition. To enhance readability, in rules we omit irrelevant elements with the place-holder $-$. We comment the rules. Rule $[\text{DCC}|_{\text{Recv}}]$ models message reception: if the queue correlating with t_c (obtained from the evaluation of expression e against the state of the receiving process) has a message on operation o_j , we remove the message from the queue and assign the payload to the variable x_j in the state of the process. Rule $[\text{DCC}|_{\text{Newque}}]$ adds to M an empty queue (ε) correlating with a key stored in x . As for BC in rule $[\text{P}|_{\text{Sup}}]$, we do not impose a structure for correlation keys, yet we require that they are distinct within their service. Rule $[\text{DCC}|_{\text{Send}}]$ models message delivery between processes in different services: the rule adds the message from the sender at the end of the correlating queue of the receiver. Rule $[\text{DCC}|_{\text{Start}}]$ accepts the creation of a new processes in a service upon request from an external process. The spawned process has B_2 as its behaviour and an empty state, except for x that stores the payload of the request.

$$\begin{array}{c}
\frac{e \downarrow_t t_c \quad j \in I \quad M(t_c) = (o_j, t_m) :: \tilde{m} \quad P = B_j \cdot t \triangleleft (x_j, t_m)}{\langle -, \&_{i \in I} ([o_i(x_i) \text{ from } e] \{B_i\}) \cdot t \mid -, M \rangle \rightarrow \langle -, P \mid -, M[t_c \mapsto \tilde{m}] \rangle} [\text{DCC}|_{\text{Recv}}] \\
\\
\frac{t_c \notin \mathbf{dom}(M)}{\langle -, \nu x; B \cdot t \mid -, M \rangle \rightarrow \langle -, B \cdot t \triangleleft (x, t_c) \mid -, M[t_c \mapsto \varepsilon] \rangle} [\text{DCC}|_{\text{Newque}}] \\
\\
\frac{P = o @_{e_1}(e_2) \text{ to } e_3; B \cdot t \quad e_1, e_2, e_3 \downarrow_t l, t_m, t_c \quad t_c \in \mathbf{dom}(M)}{\langle -, P \mid -, - \rangle \mid \langle -, M \rangle_l \rightarrow \langle -, B \cdot t \mid -, M \rangle \mid \langle -, M[t_c \mapsto M(t_c) :: (o, t_m)] \rangle_l} [\text{DCC}|_{\text{Send}}] \\
\\
\frac{Q = B_2 \cdot \emptyset \triangleleft (x, t') \quad e_1 \downarrow_t l \quad e_2 \downarrow_t t'}{\langle -, ? @_{e_1}(e_2); B_1 \cdot t \mid -, - \rangle \mid \langle !(x); B_2, -, - \rangle_l \rightarrow \langle -, B_1 \cdot t \mid -, - \rangle \mid \langle !(x); B_2, Q \mid -, - \rangle_l} [\text{DCC}|_{\text{Start}}]
\end{array}$$

Fig. 5. Dynamic Correlation Calculus, semantics (selected).

5 Compiler from FC to DCC and Properties

We now present our main result: the correct compilation of FC programs into networks of DCC services. Given a term D, C in FC, our compilation consists of three steps: (1) it projects C into a parallel composition of *endpoint choreographies*, each describing the behaviour of a single process or service in C ; (2) it encodes D to a Backend deployment; (3) it compiles the Backend choreography, result of the two previous steps, into DCC programs.

Step 1: Endpoint Projection (EPP). Given a choreography C , its EPP, denoted $\llbracket C \rrbracket$, returns an operationally-equivalent composition of endpoint choreographies. Intuitively, an endpoint choreography is a choreography that does not contain complete actions—terms (*start*) and (*com*) in FC—describing the behaviour of a process, which can be either a service process or an active one. Our definition of EPP is a straightforward adaptation of that presented in [23], so we omit it here (see [22] for the full definition). First, we define a *process projection* to derive the endpoint choreography of a single process p from a choreography C , written $\llbracket C \rrbracket_p$. Then, we formalise the EPP of a choreography as the parallel composition of (i) the projections of all active processes and (ii) the merging of all service processes accepting requests at the same location. In the definition below, we use two standard auxiliary operators: the grouping operator $\llbracket C \rrbracket_l$ returns the set of all service processes accepting requests at location l , and the merging operator $C \sqcup C'$ returns the service process whose behaviour is the merge of the behaviours of all the service processes accepting requests at the same location.

Definition 4 (Endpoint Projection). *Let C be a choreography. The endpoint projection of C , denoted $\llbracket C \rrbracket$, is:*

$$\llbracket C \rrbracket = \prod_{p \in \mathbf{fp}(C)} \llbracket C \rrbracket_p \mid \prod_l \left(\bigsqcup_{p \in \llbracket C \rrbracket_l} \llbracket C \rrbracket_p \right)$$

Example 2. As an example, let C be lines 5–8 of Example 1. Its EPP $\llbracket C \rrbracket$ is the parallel composition of the endpoint choreographies of processes c , s , and b , let them be respectively $\llbracket C \rrbracket_c$, $\llbracket C \rrbracket_s$, and $\llbracket C \rrbracket_b$, then $\llbracket C \rrbracket = \llbracket C \rrbracket_c \mid \llbracket C \rrbracket_s \mid \llbracket C \rrbracket_b$

$$\begin{aligned} \llbracket C \rrbracket_b &= \text{if } b.\text{closeTx}(cc, \text{order}) \{ & \llbracket C \rrbracket_c &= k:B \longrightarrow c[C].\{ \text{ok}(), \text{ko}() \} \\ & \quad k:b[B] \longrightarrow C.\text{ok}; k:b[B] \longrightarrow S.\text{ok} \\ \} \text{ else } \{ & \llbracket C \rrbracket_s &= k:B \longrightarrow s[S].\{ \text{ok}(), \text{ko}() \} \\ & \quad k:b[B] \longrightarrow C.\text{ko}; k:b[B] \longrightarrow S.\text{ko} \\ \} \end{aligned}$$

As shown above, the projection of the conditional is homomorphic on the process (b) that evaluates it. The projection of (*com*) terms results into a partial (*send*) for the sender — as in the two branches of the conditional in $\llbracket C \rrbracket_b$ — and a partial (*recv*) for the receiver — as in $\llbracket C \rrbracket_c$ and $\llbracket C \rrbracket_s$. Note that the EPP merges branching behaviours: in $\llbracket C \rrbracket_c$ and $\llbracket C \rrbracket_s$ the two complete communications are merged into a partial reception on either operation *ok* or *ko*.

Below, $C \prec C'$ is the standard *pruning relation* [14], a strong typed bisimilarity such that C has some unused branches and always-available accepts. In FC, the EPP preserves well-typedness and behaviour:

Theorem 3 (EPP Theorem). *Let D, C be well-typed. Then,*

1. (Well-typedness) $D, \llbracket C \rrbracket$ is well-typed.
2. (Completeness) $D, C \rightarrow D', C'$ implies $D, \llbracket C \rrbracket \rightarrow D', C''$ and $\llbracket C' \rrbracket \prec C''$.
3. (Soundness) $D, \llbracket C \rrbracket \rightarrow D', C'$ implies $D, C \rightarrow D', C''$ and $\llbracket C'' \rrbracket \prec C'$.

Step 2: Encoding to BC. After the EPP, we use our deployment encoding to obtain an operationally-equivalent system in BC. From Theorems 2 and 3 we derive Corollary 1:

Corollary 1. *Let $\Gamma \vdash D, C$. Then:*

1. (Completeness) $D, C \rightarrow D', C'$ implies $\langle D \rangle^\Gamma, \llbracket C \rrbracket \rightarrow \langle D' \rangle^{\Gamma'}, C''$ for some Γ' s.t. $\Gamma' \vdash D', C''$ and $\llbracket C' \rrbracket \prec C''$;
2. (Soundness) $\langle D \rangle^\Gamma, \llbracket C \rrbracket \rightarrow \mathcal{D}, C'$ implies $D, C \rightarrow D', C''$ for some Γ' s.t. $\Gamma' \vdash D', C''$, $\langle D' \rangle^{\Gamma'} = \mathcal{D}$ and $\llbracket C'' \rrbracket \prec C'$.

Step 3: from BC to DCC. Given $\Gamma \vdash D, C$, where C is a composition of endpoint choreographies as returned by our EPP, we define a compilation $\overline{D, C}^\Gamma$ into DCC by using the deployment encoding from FC to BC. To define $\overline{D, C}^\Gamma$, we use:

- $C|_l$, to return the endpoint choreography for location l in C (e.g., acc $k: l.p[A]; C''$);
- $C|_p$, to return the endpoint choreography of process p in C ;
- \overline{C}^Γ , to compile an endpoint choreography C to DCC, using the (selected) rules in Fig. 6;
- $l \in \Gamma$, a predicate satisfied if, according to Γ , location l contains or can spawn processes;
- $\mathcal{D}|_l$ returns the partial function of type $\mathcal{T} \rightarrow \text{Seq}(\mathcal{O} \times \mathcal{T})$ that corresponds to the projection of function g_m in \mathcal{D} with location l fixed.

Definition 5 (Compilation). *Let $\Gamma \vdash D, C$, where C is a composition of endpoint choreographies, and $\mathcal{D} = \langle D \rangle^\Gamma$. The compilation $\overline{D, C}^\Gamma$ is defined as*

$$\overline{D, C}^\Gamma = \prod_{l \in \Gamma} \left\langle \overline{C|_l}^\Gamma, \prod_{p \in \mathcal{D}(l)} \overline{C|_p}^\Gamma \cdot \mathcal{D}(p), \mathcal{D}|_l \right\rangle$$

Intuitively, for each service $\langle B_s, P, M \rangle_l$ in the compiled network: (i) the start behaviour B_s is the compilation of the endpoint choreography in C accepting the creation of processes at location l ; (ii) P is the parallel composition of the compilation of all active processes located at l , equipped with their respective states according to $\mathcal{D} = \langle D \rangle^\Gamma$; (iii) M is the set of queues in \mathcal{D} corresponding to

$$\begin{aligned}
 & \text{Let } p@l' \in \Gamma, \boxed{\text{req } k : p[A] \Leftrightarrow \bar{l}.B; C}^\Gamma = \mathbf{start}(k, l'.A, \bar{l}.B; \bar{C})^\Gamma \\
 & \mathbf{start}(k, l'.A, \bar{l}.B) = \\
 & \quad \underbrace{\bigodot_{I \in \{A, \bar{B}\}} \underline{k}.I.l = l_I}_{(s_1) \text{ store locations}} ; \underbrace{\bigodot_{I \in \{\bar{B}\}} \left(\nu \underline{k}.I.A ; ?@k.I.l(k) ; \right.}_{(s_2) \text{ correlation keys and service processes}} \\
 & \quad \left. \text{sync}(k) \text{ from } \underline{k}.I.A \right) ; \underbrace{\bigodot_{I \in \{\bar{B}\}} \text{start}@k.I.l(k) \text{ to } \underline{k}.A.I}_{(s_3) \text{ handle session start}} \\
 & \text{Let } l \in \tilde{l}, \tilde{l} : G\langle A | \tilde{C} | \tilde{D} \rangle \in \Gamma, \boxed{\text{acc } k : l.q[B]; C}^\Gamma = \mathbf{accept}(k, B, \Gamma(\tilde{l}); \bar{C})^\Gamma, \\
 & \mathbf{accept}(k, B, G\langle A | \tilde{C} | \tilde{D} \rangle) = \\
 & \quad \underbrace{!(k)}_{(a_1)} ; \underbrace{\bigodot_{I \in \{A, \tilde{C}\} \setminus \{B\}} \left(\nu \underline{k}.I.B \right)}_{(a_2)} ; \underbrace{\text{sync}@k.A.l(k) \text{ to } \underline{k}.B.A}_{(a_3)} ; \underbrace{\text{start}(k) \text{ from } \underline{k}.A.B}_{(a_4)}
 \end{aligned}$$

Fig. 6. Compiler from Endpoint Choreographies to DCC.

location l . We comment the rules in Fig. 6, where the notation \bigodot is the sequence of behaviours $\bigodot_{i \in [1, n]} (B_i) = B_1; \dots; B_n$.

Requests. Function \mathbf{start} defines the compilation of (*req*) terms: it compiles (*req*) terms to create the queues and a part of the session descriptor of a valid session support (mirroring rule $[\mathcal{P}]_{\text{sup}}$) for the starter. Given a session identifier k , the located role of the starter ($l'.A$), and the other located roles in the session ($\bar{l}.B$), function \mathbf{start} returns DCC code that: (s_1) includes in the session descriptor all the locations of the processes involved in the session. In (s_2) it adds all the keys correlating with the queues of the starter for the session, it requests the creation of all the service processes for the session, and it waits for them to be ready using the reserved operation sync . Finally, (s_3) it sends to them the complete session descriptor obtained after the reception (in the sync step) of all correlation keys from all processes.

Accepts. Term (*acc*) defines the start behaviour of a spawned process at a location. Given a session identifier k , the role B of the service process, and the service typing $G\langle A | \tilde{C} | \tilde{D} \rangle$ of the location, function \mathbf{accept} compiles the code that: (a_1) accepts the request to spawn a process, (a_2) creates its queues and keys, updates the session descriptor received from the starter, and sends it back to the latter (a_3). Finally with (a_4) the new process waits to start the session.

Example 3. We compile the first two lines of the choreography C in Example 1.

$$\begin{aligned}
 & \boxed{D, \bar{C}}^\Gamma = \langle \mathbf{0}, P_c \rangle_{l_c} \mid \langle B_s, \mathbf{0} \rangle_{l_s} \mid \langle B_b, \mathbf{0} \rangle_{l_b} \\
 & \text{where } P_c = \begin{cases} \underline{k}.S.l = l_s; \underline{k}.B.l = l_b; \nu \underline{k}.S.C; ?@k.S.l(k); \text{sync}(k) \text{ from } \underline{k}.S.C; \nu \underline{k}.B.C; ?@k.B.l(k); \\ \text{sync}(k) \text{ from } \underline{k}.B.C; \text{start}@k.S.l(k) \text{ to } \underline{k}.C.S; \text{start}@k.B.l(k) \text{ to } \underline{k}.C.B; \\ /* \text{ end of start-request */ } \text{buy}@k.S.l(\text{product}) \text{ to } \underline{k}.C.S; \dots \end{cases} \\
 & \text{and } B_s = \begin{cases} !(k); \nu \underline{k}.C.S; \nu \underline{k}.B.S; \text{sync}@k.C.l(k) \text{ to } \underline{k}.S.C; \text{start}(k) \text{ from } \underline{k}.C.S; \\ /* \text{ end of accept */ } \text{buy}(x) \text{ from } \underline{k}.C.S; \dots \end{cases}
 \end{aligned}$$

We omit to report B_b , which is similar to B_s .

Properties. We report the main properties of our compilation to DCC.

In our definition, we use the term *projectable* to indicate that, given a choreography C , we can obtain its projection $\llbracket C \rrbracket$. Theorem 4 defines our result, for which, given a well-typed, projectable Frontend choreography, we can obtain its correct implementation as a DCC network.

Theorem 4 (Applied Choreographies). *Let D, C be a Frontend choreography where C is projectable and $\Gamma \vdash D, C$ for some Γ . Then:*

1. (Completeness) $D, C \rightarrow D', C'$ implies $\boxed{\llbracket D \rrbracket^\Gamma, \llbracket C \rrbracket}^\Gamma \rightarrow^+ \boxed{\langle D' \rangle^{\Gamma'}, C''}^{\Gamma'}$ and $\llbracket C' \rrbracket \prec C''$ and for some $\Gamma', \Gamma' \vdash D', C'$.
2. (Soundness) $\boxed{\llbracket D \rrbracket^\Gamma, \llbracket C \rrbracket}^\Gamma \rightarrow^* S$ implies $D, C \rightarrow^* D', C'$ and $S \rightarrow^* \boxed{\langle D' \rangle^{\Gamma'}, C''}^{\Gamma'}$ and $\llbracket C' \rrbracket \prec C''$ and for some $\Gamma', \Gamma' \vdash D', C'$.

By Theorems 1 and 4, deadlock-freedom is preserved from well-typed choreographies to their final translation in DCC. We say that a network S in DCC is deadlock-free if it is either a composition of services with terminated running processes or it can reduce.

Corollary 2. $\Gamma \vdash D, C$ and $\text{co}(\Gamma)$ imply that $\boxed{D, \llbracket C \rrbracket}^\Gamma$ is deadlock-free.

6 Related Work and Discussion

Choreography Languages. This is the first correctness result of an end-to-end translation from choreographies to an abstract model based on a real-world communication mechanism. Previous formal choreography languages specify only an EPP procedure towards a calculus based on name synchronisation, leaving the design of its concrete support to implementors. Chor and AIOCJ [11, 29] are the respective implementations of the models found in [2, 8]. However, the implementation of their EPP significantly departs from their respective formalisation, since the former are based on message correlation. This gap breaks the correctness-by-construction guarantee of choreographies—there is no proof that the implementation correctly supports synchronisation on names. Implementations of other frameworks based on sessions share similar issues. For example, Scribble [7] is a protocol definition language based on multiparty asynchronous session types [13] used to statically [30] and dynamically [31, 32] check compliance of interacting programs. Our work can be a useful reference to formalise the implementation of these session-based languages.

Delegation. Delegation supports the transferring of the responsibility to continue a session from a process to another [13] and it was introduced to choreographies in [2]. Introducing delegation in FC is straightforward, since we can just import the development from [2]. Implementing it in BC and DCC would be more involved, but not difficult: delegating a role in a session translates to moving the content of a queue from a process to another, and ensuring that future messages

reach the new process. The mechanisms to achieve the latter part have been investigated in [30], which would be interesting to formalise in our framework.

Correlation keys. In the semantics of BC, we abstract from how correlation keys are generated. With this loose definition we capture several implementations, provided they satisfy the requirement of uniqueness of keys (wrt to locations). As future work, we plan to implement a language, based on our framework, able to support custom procedures for the generation of correlation keys (e.g., from database queries, cookies, etc.).

Acknowledgements. This work was partially supported by the Independent Research Fund Denmark, grant no. DFF-7014-00041.

References

1. Montesi, F.: Choreographic Programming. Ph.D. thesis, IT University of Copenhagen (2013)
2. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: POPL, pp. 263–274. ACM (2013)
3. W3C WS-CDL Working Group: WS-CDL version 1.0 (2004). <http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/>
4. OMG: Business Process Model and Notation. <http://www.omg.org/spec/BPMN/2.0/>
5. Lanese, I., Guidi, C., Montesi, F., Zavattaro, G.: Bridging the gap between interaction - and process-oriented choreographies. In: SEFM, pp. 323–332. IEEE (2008)
6. Basu, S., Bultan, T., Ouederni, M.: Deciding choreography realizability. In: POPL, pp. 191–202. ACM (2012)
7. Honda, K., Mukhamedov, A., Brown, G., Chen, T.-C., Yoshida, N.: Scribbling interactions with a formal foundation. In: Natarajan, R., Ojo, A. (eds.) ICDCIT 2011. LNCS, vol. 6536, pp. 55–75. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19056-8_4
8. Dalla Preda, M., Gabbriellini, M., Giallorenzo, S., Lanese, I., Mauro, J.: Dynamic choreographies. In: Holvoet, T., Viroli, M. (eds.) COORDINATION 2015. LNCS, vol. 9037, pp. 67–82. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19282-6_5
9. Pi4soa (2008). <http://www.pi4soa.org>
10. JBoss Community: Savara. <http://www.jboss.org/savara/>
11. Chor Programming Language. <http://www.chor-lang.org/>
12. AIOCJ framework. <http://www.cs.unibo.it/projects/jolie/aioj.html>
13. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM (JACM) **63**(1), 9 (2016)
14. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for web services. ACM Trans. Program. Lang. Syst. **34**(2), 8:1–8:78 (2012)
15. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. Distrib. Comput. **31**(1), 51–67 (2018)
16. Carbone, M., Montesi, F., Schürmann, C., Yoshida, N.: Multiparty session types as coherence proofs. Acta Inf. **54**(3), 243–269 (2017)

17. Qiu, Z., Zhao, X., Cai, C., Yang, H.: Towards the theoretical foundation of choreography. In: WWW, pp. 973–982. IEEE Computer Society Press (2007)
18. Milner, R. (ed.): A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980). <https://doi.org/10.1007/3-540-10235-3>
19. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I and II. *Inf. Comput.* **100**, 1–40, 41–77 (1992)
20. Giallorenzo, S.: Real-World Choreographies. Ph.D. thesis, University of Bologna, Italy (2016)
21. OASIS: WS-BPEL (2007). <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>
22. Gabbrielli, M., Giallorenzo, S., Montesi, F.: Applied choreographies. Technical report (2018). http://www.saveriogiallorenzo.com/publications/AC/AC_tr.pdf
23. Montesi, F., Yoshida, N.: Compositional choreographies. In: D’Argenio, P.R., Mellgratti, H. (eds.) CONCUR 2013. LNCS, vol. 8052, pp. 425–439. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40184-8_30
24. Sangiorgi, D., Walker, D.: The π -Calculus: A Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
25. Pierce, B.C.: Types and Programming Languages. MIT Press, Cambridge (2002)
26. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. *Math. Struct. Comput. Sci.* **26**(2), 238–302 (2016)
27. Montesi, F., Carbone, M.: Programming services with correlation sets. In: Kappel, G., Maamar, Z., Motaahari-Nezhad, H.R. (eds.) ICSOC 2011. LNCS, vol. 7084, pp. 125–141. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25535-9_9
28. Montesi, F., Guidi, C., Zavattaro, G.: Service-oriented programming with Jolie. In: Bouguettaya, A., Sheng, Q., Daniel, F. (eds.) Web Services Foundations, pp. 81–107. Springer, New York (2014). https://doi.org/10.1007/978-1-4614-7518-7_4
29. Dalla Preda, M., Giallorenzo, S., Lanese, I., Mauro, J., Gabbrielli, M.: AIOCJ: a choreographic framework for safe adaptive distributed applications. In: Combe-male, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 161–170. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11245-9_9
30. Hu, R., Yoshida, N., Honda, K.: Session-based distributed programming in Java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 516–541. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70592-5_22
31. Demangeon, R., Honda, K., Hu, R., Neykova, R., Yoshida, N.: Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. *Formal Methods Syst. Des.* **46**(3), 197–225 (2015)
32. Neykova, R., Yoshida, N.: Multiparty session actors. *Logical Methods Comput. Sci.* **13**(1) (2017)



Monotonic Prefix Consistency in Distributed Systems

Alain Girault¹, Gregor Gössler¹, Rachid Guerraoui², Jad Hamza^{3(✉)},
and Dragos-Adrian Seredinschi²

¹ Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG,
38000 Grenoble, France

{alain.girault,gregor.goessler}@inria.fr

² LPD, EPFL, Lausanne, Switzerland

{rachid.guerraoui,dragos-adrian.seredinschi}@epfl.ch

³ LARA, EPFL, Lausanne, Switzerland

jad.hamza@epfl.ch

Abstract. We study the issue of data consistency in distributed systems. Specifically, we consider a distributed system that replicates its data at multiple sites, which is prone to partitions, and which is assumed to be available (in the sense that queries are always eventually answered). In such a setting, strong consistency, where all replicas of the system apply synchronously every operation, is not possible to implement. However, many weaker consistency criteria that allow a greater number of behaviors than strong consistency, are implementable in available distributed systems.

We focus on determining the strongest consistency criterion that can be implemented in a convergent and available distributed system that tolerates partitions. We focus on objects where the set of operations can be split into updates and queries. We show that no criterion stronger than Monotonic Prefix Consistency (MPC) can be implemented.

1 Introduction

Replication is a mechanism that enables sites from different geographical locations to access a shared data type with low latency. It consists of creating copies of this data type on each *site* of a distributed system. Ideally, replication should be transparent, in the sense that the users of the data type should not notice discrepancies between the different copies of the data type.

An ideal replication scheme could be implemented by keeping all sites synchronized after each update to the data type. This ideal model is called *strong consistency*, or linearizability [1]. The disadvantage of this model is that it can cause large delays for users, and worse the data type might not be *available* to use at all times. This may happen, for instance, if some sites of the system are unreachable, i.e., partitioned from the rest of the network. Briefly, it is not possible to implement strong consistency in a distributed system while ensuring *high availability* [2–4].

High availability (hereafter *availability* for short) means that sites must answer users' requests directly, without waiting for outside communication.

Given this impossibility, developers rely on weaker notions of consistency, such as causal consistency [5]. Weaker consistency criteria do not require sites to be exactly synchronized as in strong consistency. For instance, causal consistency allows different sites to apply updates to the data type in different orders, as long as the updates are not *causally related*. Informally, a consistency criterion specifies the *behaviors* that are allowed by a replicated data type. In this sense, causal consistency is more permissive than strong consistency. We also say that strong consistency is *stronger* than causal consistency, as strong consistency allows strictly fewer behaviors than causal consistency. A natural question is then: What is the strongest consistency criterion that can be implemented by a replicated data type? We focus in this paper on data types where the set of operations can be split into two disjoint sets, updates and queries. Updates modify the state and but do not return values, while queries return values without modifying the state.

In [4], it was proven that nothing stronger than *observable causal consistency* (a variant of causal consistency) can be implemented. It is an open question whether observable causal consistency itself is actually implementable. Moreover, [4] does not study consistency criteria that are not comparable to observable causal consistency. Indeed, there exist consistency criteria that are neither stronger than causal consistency, nor weaker, and which can be implemented by a replicated data type.

In our paper, we explore one such consistency criterion. More precisely, we prove that, under some conditions which are natural in a large distributed system (availability and convergence), nothing stronger than *monotonic prefix consistency* (MPC) [6] can be implemented. This result does not contradict the result from [4], since MPC and causal consistency are incomparable.

The reason why MPC and observable causal consistency are incomparable is as follows. MPC requires all sites to apply updates in the same order (but not necessarily synchronized at the same time, as in strong consistency), while causal consistency allows non-causally related updates to be applied in different orders. On the other hand, causal consistency requires all causally-related updates to be applied in an order respecting causality, while MPC requires no such constraint.

Overall, our contribution is to prove that, for a notion of behaviors where the time and place of origin of updates do not matter, nothing stronger than MPC can be implemented in a distributed setting. Moreover, we remark that clients that only have the observability defined in Sect. 3 cannot tell the difference between a strongly consistent implementation and an MPC implementation.

In the rest of this paper, we first give preliminary notions and a formal definition of the problem we are addressing (Sects. 2 and 3). We then turn our attention to the MPC model by defining it formally and through an implementation (Sect. 4). We prove that, given the observability mentioned above, and under conditions natural in a large-scale network (availability, convergence), nothing

stronger than MPC can be implemented (Sect. 6). Then we compare MPC with other consistency models (Sect. 7), and conclude (Sect. 8).

To improve the presentation, some proofs are deferred to the appendix.

2 Replicated Implementations

An *implementation* of a replicated data type consists of several *sites* that communicate by sending messages. Messages are delivered asynchronously by the network, and can be reordered or delayed. To be able to build implementations that provide liveness guarantees, we assume all messages are *eventually* delivered by the network.

Each site of an implementation maintains a local state. This local state reflects the view that the site has on the replicated data type, and may contain arbitrary data. Each site implements the protocol by means of an *update handler*, a *query handler*, and a *message handler*.

The update handler is used by (hypothetical) clients to submit updates to the data type. The update handler may modify the local states of the site, and broadcast a message to the other sites. Later, when another site receives the message, its *message handler* is triggered, possibly updating the local state of the site, and possibly broadcasting a new message.

The *query handler* is used by clients to make queries on the data type. The query handler returns an answer to the client, and is a read-only operation that does not modify the local state or broadcast messages.

Remark 1. Our model only supports broadcast and not general peer-to-peer communication, but this is without loss of generality. We can simulate sending a message to a particular site by writing the identifier of the receiving site in the broadcast message. All other sites would then simply ignore messages that are not addressed to them.

In this paper, we consider implementations of the *list data type*. The list supports an update operation of the form $\mathbf{write}(d)$, with $d \in \mathbb{N}$, which adds the element d to the end of the list. The list also supports a query operation \mathbf{read} that returns the whole list $\ell \in \mathbb{N}^*$, which is a sequence of elements in \mathbb{N} .

Definition 1. Let $\mathbf{Upd} = \{\mathbf{write}(d) \mid d \in \mathbb{N}\}$ be the set of updates, and $\mathbf{Ans} = \{\mathbf{read}(\ell) \mid \ell \in \mathbb{N}^*\}$ be the set of all possible answers to queries.

We focus on the list data type because queries return the history of all updates that ever happened. In that regard, lists can encode any other data type whose operations can be split in updates and queries, by adding a processing layer after the query operation of the list returns all updates. Data types that contain operations which are queries and updates at the same time (e.g. the Pop operation of a stack) are outside the scope of this paper. We now proceed to give the formal syntax for implementations, and then the corresponding operational semantics.

Definition 2. An implementation \mathcal{I} is a tuple $(Q, \iota, \mathbb{P}, \mathbf{Msg}, \mathbf{msg_handler}, \mathbf{update_handler}, \mathbf{query_handler})$ where

- Q is a non-empty set of local states,
- \mathbb{P} is a non-empty finite set of process identifiers,
- $\iota : \mathbb{P} \rightarrow Q$ associates to each process an initial local state,
- Msg is a set of messages,
- $\text{msg_handler} : Q \times \text{Msg} \rightarrow Q \times \text{Msg}^\perp$ is a total function, called the handler of incoming messages, which updates the local state of a site when a message is received, and possibly broadcasts a new message,
- $\text{update_handler} : Q \times \text{Upd} \rightarrow Q \times \text{Msg}^\perp$ is a total function, called the handler of updates, which modifies the local state when an update is submitted, and possibly broadcasts a message.
- $\text{query_handler} : Q \rightarrow \text{Ans}$ is a total function, called the handler of client queries, which returns an answer to client queries.

The set Msg^\perp is defined as $\text{Msg} \uplus \{\perp\}$, where \perp is a special symbol denoting the fact that no message is sent.

Before defining the semantics of implementations, we introduce a few notations. We first define the notion of an *action*, used to denote events that happen during the execution. Each action contains a unique *action identifier* $\text{aid} \in \mathbb{N}$, and the process identifier $\text{pid} \in \mathbb{P}$ where the action occurs.

Definition 3. A broadcast action is a tuple $(\text{aid}, \text{pid}, \text{broadcast}(\text{mid}, \text{msg}))$, and a receive action is a tuple $(\text{aid}, \text{pid}, \text{receive}(\text{mid}, \text{msg}))$, where $\text{mid} \in \mathbb{N}$ is the message identifier and $\text{msg} \in \text{Msg}$ is the message. An update action or a write action is a tuple $(\text{aid}, \text{pid}, \text{write}(d))$ where $d \in \mathbb{N}$. Finally, a query action or a read action is a tuple $(\text{aid}, \text{pid}, \text{read}(\ell))$ where $\ell \in \mathbb{N}^*$.

Executions are then defined as sequences of actions, and are considered up to action and message identifiers renaming.

Definition 4. An execution e is a sequence of broadcast, receive, query and update actions where no two actions have the same identifier aid , and no two broadcast actions have the same message identifier mid .

We now describe how implementations operate on a given site $\text{pid} \in \mathbb{P}$.

Definition 5. We say that a sequence of actions $\sigma_1 \dots \sigma_n \dots$ from site pid follows \mathcal{I} if there exists a sequence of states $q_0 \dots q_n \dots$ such that $q_0 = \iota(\text{pid})$, and for all $i \in \mathbb{N} \setminus \{0\}$, we have:

1. if $\sigma_i = (\text{aid}, \text{pid}, \text{write}(d))$ with $d \in \mathbb{N}$, then $\text{update_handler}(q_{i-1}, \text{write}(d)) = (q_i, _)$. This means that upon a write action, a site must update its state as defined by the update handler;
2. if $\sigma_i = (\text{aid}, \text{pid}, \text{read}(\ell))$ with $\ell \in \mathbb{N}^*$, then $\text{query_handler}(q_{i-1}) = \text{read}(\ell)$ and $q_i = q_{i-1}$. This condition states that query actions do not modify the state, and that the answer $\text{read}(\ell)$ given to query actions must be as specified by the query handler, depending on the current state q_{i-1} ;
3. if $\sigma_i = (\text{aid}, \text{pid}, \text{broadcast}(\text{mid}, \text{msg}))$, then $q_i = q_{i-1}$. Broadcast actions do not modify the local state;

4. if $\sigma_i = (aid, pid, \text{receive}(mid, msg))$, then $\text{msg_handler}(q_{i-1}, msg) = (q_i, -)$.
 The reception of a message modifies the local state as specified by `msg_handler`.

Moreover, we require that broadcast actions are performed if and only if they are triggered by the handler of incoming messages, or the handler of clients requests. Formally, for all $i > 0$, $\sigma_i = (aid, pid, \text{broadcast}(mid, msg))$ if and only if either:

5. $\exists \text{write}(d) \in \text{Upd}$ and $aid' \in \mathbb{N}$ such that $\sigma_{i-1} = (aid', pid, \text{write}(d))$ and $\text{update_handler}(q_{i-1}, \text{write}(d)) = (q_i, msg)$, or
 6. $\exists aid' \in \mathbb{N}$, $mid \in \mathbb{N}$, and $msg' \in \text{Msg}$ such that
 $\sigma_{i-1} = (aid', pid, \text{receive}(mid, msg))$ and $\text{msg_handler}(q_{i-1}, msg') = (q_i, msg)$.

When all conditions hold, we say that $q_0 \dots q_n \dots$ is a run for $\sigma_1 \dots \sigma_n \dots$. Note that when a run exists for a sequence of actions, it is unique.

We then define the set of executions generated by \mathcal{I} , denoted $\llbracket \mathcal{I} \rrbracket$. In particular, this definition models the communication between sites, and specifies that a receive action may happen only if there exists a broadcast action with the same message identifier preceding the receive action in the execution. Moreover, a *fairness* condition ensures that, in an infinite execution, every broadcast action must have a corresponding receive action on every site.

Definition 6. Let \mathcal{I} be an implementation. The set of executions generated by \mathcal{I} is $\llbracket \mathcal{I} \rrbracket$ such that $e \in \llbracket \mathcal{I} \rrbracket$ if and only if the three following conditions hold:

- **Projection:** for all $pid \in \mathbb{P}$, the projection $e|_{pid}$ follows \mathcal{I} ,
- **Causality:** for every receive action $\sigma = (aid, pid, \text{receive}(mid, msg))$, there exists a broadcast action $(aid', pid', \text{broadcast}(mid, msg))$ before σ in e ,
- **Fairness:** if e is infinite, then for every site $pid \in \text{Pid}$ and every broadcast action $(aid', pid', \text{broadcast}(mid, msg))$ performed on any site pid' , there exists a receive action $(aid, pid, \text{receive}(mid, msg))$ in e ,

where $e|_{pid}$ is the subsequence of e of actions performed by process pid :

- $\varepsilon|_{pid} = \varepsilon$;
- $((aid, pid, x).e)|_{pid} = (aid, pid, x).(e|_{pid})$;
- $((aid, pid', x).e)|_{pid} = e|_{pid}$ whenever $pid' \neq pid$.

Remark 2. The implementations we consider are *available* by construction, in the sense that any site allows any updates or queries to be done at any time, and answers to queries directly. This is ensured by the fact that our update and query handlers are total functions. More precisely, the item 1 of Definition 5 (together with Definition 6) ensures that updates can be performed at any time through the update handler (*update availability*).

The broadcast action that happens right after an update action must be thought of as happening right after the update. Broadcast actions do not involve actively waiting for responses, and as such do not prevent availability.

Similarly, the item 2 of Definition 5 ensures that any query of any site is answered immediately, only using the local state of the site (*query availability*). We later formalize this in Lemmas 1 and 2.

For the rest of the paper, we consider that updates are unique, in the sense that an execution may not contain two update actions that write the same value $d \in \mathbb{N}$. This assumption only serves to simplify the presentation of our result, and can be done without loss of generality, as updates can be made unique by attaching a unique timestamp to them.

3 Problem Definition

In this section, we explain how we compare implementations using the notion of a *trace*. Informally, the trace of an execution corresponds to what is observable from the point of view of clients using the data type.

Our notion of a trace is based on two assumptions: (1) Clients know the order of the queries they have done on a site, but not the relative positions of their queries with respect to other clients' queries. (2) The origin of updates is not relevant from a client's perspective. This models publicly accessible data structures where any client can disseminate a transaction in the network, and the place and time where the transaction was created are not relevant for the protocol execution.

More precisely, a trace records an unordered set of updates (without their site identifiers), and records for each site the sequence of queries that happened on this site.

Definition 7. A trace (t_r, W) is a pair where t_r is a labelled partially ordered set (see hereafter for more details), and W is a subset of \mathbb{N} . The trace (t_r, W) corresponding to an execution e is denoted $\text{tr}(e)$, where $t_r = (A, <, \text{label})$ is a labelled partially ordered set such that:

- A is the set of action identifiers of query actions of e ;
- $<$ is a transitive and irreflexive relation over A , sometimes called the program order, ordering queries performed on the same site; more precisely, we have $\text{aid} < \text{aid}'$ if $\text{aid}, \text{aid}' \in A$ are action identifiers performed by the same site, and that appear in that order in e ;
- $\text{label} : A \rightarrow \text{Ans}$ is the labelling function such that for any $\text{aid} \in A$, $\text{label}(\text{aid})$ is the answer of the query action corresponding to aid in e ;

and $W \subseteq \mathbb{N}$ is the set of elements that appear in an update action of e .

Example 1. Consider the execution e in Fig. 1, and its corresponding trace $\text{tr}(e)$. ($\text{pid}_1, \text{pid}_2, \text{pid}_3 \in \mathbb{P}$ are site identifiers, $\text{mid}_1, \text{mid}_2, \text{mid}_3 \in \mathbb{N}$ are unique message identifiers, and $\text{msg}_1, \text{msg}_2, \text{msg}_3 \in \text{Msg}$ are messages).

Then, we compare implementations by looking at the set of traces they produce. The fewer traces an implementation produces, the stronger it is, and the closer it is to strong consistency.

Definition 8. The notation $\text{tr}()$ is extended to sets of executions point-wise. An implementation \mathcal{I}_1 is stronger than \mathcal{I}_2 , denoted $\mathcal{I}_1 \preceq \mathcal{I}_2$ iff

$$\text{tr}(\llbracket \mathcal{I}_1 \rrbracket) \subseteq \text{tr}(\llbracket \mathcal{I}_2 \rrbracket)$$

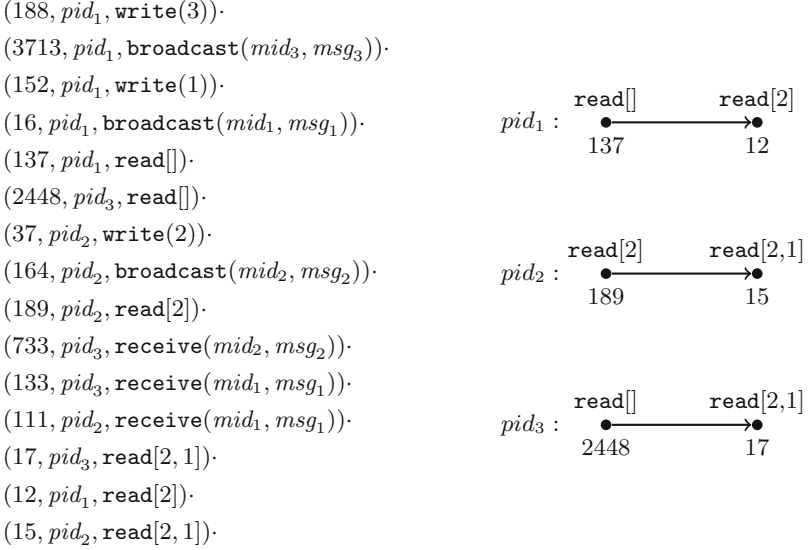


Fig. 1. An execution e read from top to bottom, then left to right (188, 3713, \dots , 189, 733, \dots , 15) and its corresponding trace $\text{tr}(e) = (t_r, W)$ (right). The bullets represent the action identifiers of t_r (written under the bullet), and the corresponding labels are represented right above. The arrows represent the program order $<$ of t_r . The set of writes is W is $\{1, 2, 3\}$ (from actions 152, 37, and 188 respectively).

The implementations \mathcal{I}_1 and \mathcal{I}_2 are said to be equivalent, denoted $\mathcal{I}_1 \approx \mathcal{I}_2$, iff $\mathcal{I}_1 \preceq \mathcal{I}_2$ and $\mathcal{I}_2 \preceq \mathcal{I}_1$. Moreover, \mathcal{I}_1 is strictly stronger than \mathcal{I}_2 , denoted $\mathcal{I}_1 \prec \mathcal{I}_2$, iff $\mathcal{I}_1 \preceq \mathcal{I}_2$ and $\mathcal{I}_1 \not\approx \mathcal{I}_2$.

Our goal is to find an implementation \mathcal{I} which is minimal in the \preceq ordering, i.e., for which there does not exist an implementation \mathcal{I}' strictly stronger than \mathcal{I} .

4 Definition of Monotonic Prefix Consistency (MPC)

Often called consistent prefix [6, 7], the MPC model requires that all sites of the replicated system agree on the order of write operations (i.e., updates on the state). More precisely, this means that given two read operations (possibly on two different sites), one read has to return a list of writes which is a prefix of the other. Moreover, read operations which execute on the same site are monotonic. This means that subsequent reads at the same site reflect a non-decreasing prefix of writes, i.e., the prefix must either increase or remain unchanged. The trace given in Fig. 1 satisfies these constraints.

Note that the order on write operations on which the sites agree does not necessarily satisfy causality among these operations nor real-time. In other words, the order in which clients submit write operations does not translate into any

constraints on the order in which these updates apply at all sites. Moreover, MPC does not guarantee that a read operation will return *all* of the preceding writes, only a prefix of these writes. For instance, some sites can be later than other sites in applying some updates.

Definition 9. *Given two lists $\ell_1, \ell_2 \in \mathbb{N}^*$, we say that ℓ_1 is a prefix of ℓ_2 , denoted $\ell_1 \sqsubseteq \ell_2$, if there exists $\ell_3 \in \mathbb{N}^*$ such that $\ell_2 = \ell_1 \cdot \ell_3$. Moreover, ℓ_1 is a strict prefix of ℓ_2 , denoted $\ell_1 \sqsubset \ell_2$, if $\ell_1 \sqsubseteq \ell_2$ and $\ell_1 \neq \ell_2$.*

By abuse of notation, we extend the prefix order to elements of Ans , which are of the form $\text{read}(\ell)$ where ℓ is a list (see Definition 1). Moreover, we also use the prefix notations for other types of sequences, such as executions. We now formally define MPC.

Definition 10. *MPC is the set of traces (t_r, W) where $t_r = (A, <, \text{label})$ satisfying the following conditions:*

- **Monotonicity:** *A query aid' done after aid on the same site cannot return a smaller list. For all $\text{aid}, \text{aid}' \in A$, if $\text{aid} < \text{aid}'$, then $\text{label}(\text{aid}) \sqsubseteq \text{label}(\text{aid}')$.*
- **Prefix:** *Queries done on different sites are compatible, in the sense that one is a prefix of the other. For any all $\text{aid}, \text{aid}' \in A$, $\text{label}(\text{aid}) \sqsubseteq \text{label}(\text{aid}')$ or $\text{label}(\text{aid}') \sqsubseteq \text{label}(\text{aid})$.*
- **Consistency:** *Queries only return elements that come from a write. For all $\text{aid} \in A$, and for any element $d \in \mathbb{N}$ of $\text{label}(\text{aid})$, we have $d \in W$.*

5 Feasibility of MPC

In this section, we provide a toy implementation (Fig. 2) whose traces are all in MPC, to show that MPC is indeed implementable. The idea is to let Site 1 decide on the order of all update operations. In general, the consensus mechanism for implementing MPC can be arbitrary, and symmetric with respect to sites, but we present this one for its simplicity.

For ease of presentation, we assume here that update and message handlers can be different depending on the site. This can be simulated in our original definition by using the ι function (Definition 2, Sect. 2), which defines a particular initial state for each site.

Each site maintains a local state (in Q) which is the prefix of updates as decided by Site 1. Upon receiving an update (line 16), Site i with $i > 1$ forwards the update to Site 1. When receiving an update (line 12) or when receiving a forwarded message (line 20), Site 1 updates its local state, and broadcasts an **Apply** messages for the other sites. Finally, when receiving an **Apply** messages (line 25), Site i with $i > 1$, updates its local state.

We assume that the **Apply** messages sent by Site 1 are received in the same order in which they are sent, which can be implemented by having Site 1 add a local version number to each broadcast message, and having sites with $i > 1$ cache messages until all previous messages have been received. Similarly, we

```

1 // Each site stores an element of Q, defined as a list of numbers
2 type Q = List [Nat]
3
4 abstract class Msg
5 // Forwarded messages go from Site i to Site 1, for all i > 1
6 case class Forwarded(d: Nat) extends Msg
7 // Apply messages originate from Site 1 and go to Site i, for i > 1
8 case class Apply(d: Nat) extends Msg
9
10 // The update handler for Site 1 appends element 'upd' to q,
11 // and tells the other sites to do the same with Apply(upd)
12 def update_handler(q: Q, upd: Upd) = (append(q,upd), Apply(upd))
13
14 // The update handler for Site i > 1 sends a message Forwarded(upd)
15 // which is destined for Site 1, and does not modify the state
16 def update_handler(q: Q, upd: Upd) = (q, Forwarded(upd))
17
18 // Message handler for Site 1 (ignores Apply messages)
19 def msg_handler(msg: Msg) = msg match {
20   case Forwarded(d) => (append(q,d), Apply(upd))
21 }
22
23 // Message handler for Site i > 1 (ignores Forwarded messages)
24 def msg_handler(msg: Msg) = msg match {
25   case Apply(d) => (append(q,d), ⊥)
26 }
27
28 // The query handler of any site returns the local state
29 def query_handler(q: Q) = q

```

Fig. 2. An implementation of MPC which is centralized at Site 1.

assume that each message which is sent by a site is treat at most once by each of the other sites. We omit these details in Fig. 2. Finally, the query handler of each site (line 29) simply returns the list maintained in the local state.

We now prove that all the traces of the implementation described in Fig. 2 satisfy MPC.

Proposition 1. *Let \mathcal{I} be the implementation of Fig. 2. Then $\mathcal{I} \preceq \text{MPC}$.*

The formal proof is in Appendix A. It relies on the observation that the implementation maintains the following invariant:

- (Related to **Monotonicity**) The list maintained in the local state Q of each site grows over time.
- (Related to **Prefix**) At any moment, given two lists ℓ_1 and ℓ_2 of two sites, ℓ_1 is a prefix of ℓ_2 or vice versa. Any list is always a prefix of (or equal to) the list of Site 1.
- (Related to **Consistency**) The list of a site only contains values that come from some update.

6 Nothing Stronger Than MPC in a Distributed Setting

We now proceed to our main result, stating that there exists no *convergent* implementation stronger than MPC. Convergent in our setting means that every write action performed should *eventually* be taken into account by all sites. We

formalize this notion in Sect. 6.1. This convergence assumption prevents trivial implementations, for instances ones that do not communicate and always return the empty list for all queries.

In Sect. 6.2, we prove several lemmas that hold for all implementations. We make use of these lemmas to prove our main theorem in Sect. 6.3.

6.1 Convergence Property

Convergence is formalized using the notion of eventual consistency (see e.g. [8, 9] for definitions similar to the one we use there). A trace is eventually consistent if every write is *eventually* propagated to all sites. More precisely, for every action $\mathbf{write}(d)$, the number of queries that do not contain d in their list must be finite. Note that this implies that all finite traces are eventually consistent.

Definition 11. *A trace (t_r, W) with $t_r = (A, <, \mathit{label})$ is eventually consistent if for every $d \in W$, the set $\{aid \in A \mid d \notin \mathit{label}(aid)\}$ is finite. An implementation is convergent if all of its traces are eventually consistent.*

6.2 Properties of Implementations

Lemmas 1, 2, and 3 describe basic closure properties of the set of executions generated by implementations in our setting. The semantics described in Sect. 2 ensures that new updates and queries can always be performed following an existing execution. Moreover, queries never modify the state, and therefore removing a read action from an execution does not affect its validity (Lemma 3).

Lemma 1 (Update Availability). *Let \mathcal{I} be an implementation. Let e be a finite execution in $\llbracket \mathcal{I} \rrbracket$, and let $(t_r, W) = \mathit{tr}(e)$. Let $d \in \mathbb{N}$. Then, there exists an execution $e' \in \llbracket \mathcal{I} \rrbracket$ such that e is a prefix of e' and $\mathit{tr}(e') = (t_r, W \cup \{d\})$.*

Proof. Since $e \in \llbracket \mathcal{I} \rrbracket$, we know by Definitions 5 and 6 that $e|_{pid}$ follows \mathcal{I} and that there exists a run q_0, \dots, q_n for $e|_{pid}$. Let $(q_{n+1}, \mathit{msg}) = \mathit{update_handler}(q_n, \mathbf{write}(d))$. We distinguish two cases:

- (1) If $\mathit{msg} = \perp$, let $e' = e \cdot (aid, pid, \mathbf{write}(d))$, where $aid \in \mathbb{N}$ is a fresh action identifier that does not appear in e , and pid is any process identifier in \mathbb{P} .
- (2) If $\mathit{msg} \in \mathit{Msg}$, let $e' = e \cdot (aid_1, pid, \mathbf{write}(d)) \cdot (aid_2, \mathbf{broadcast}(mid, \mathit{msg}))$, where aid_1, aid_2 are fresh action identifiers, and mid is a fresh message identifier.

In both cases, we construct a new run by adding the state q_{n+1} at the end of the run q_0, \dots, q_n (once in case 1, and twice in case 2). By Definition 5, this ensures that $e'|_{pid}$ follows \mathcal{I} , and we then obtain $e' \in \llbracket \mathcal{I} \rrbracket$ by Definition 6. Moreover, we have $\mathit{tr}(e') = (t_r, W \cup \{d\})$, which concludes our proof. \square

The next lemma shows that the implementation is *available for queries*. This means that given a finite execution, we can perform a query on any site and obtain an answer, as ensured by the definitions given in Sect. 2. The proof is in Appendix B.

Lemma 2 (Query Availability). *Let \mathcal{I} be an implementation. Let $e \in \llbracket \mathcal{I} \rrbracket$ be a finite execution and $pid \in \mathbb{P}$. Then, there exist $aid \in \mathbb{N}$ and $\ell \in \mathbb{N}^*$ such that the execution $e' = e \cdot (aid, pid, \text{read}(\ell))$ belongs to $\llbracket \mathcal{I} \rrbracket$.*

We then prove it is possible to remove any query action from an execution.

Lemma 3 (Invisible Reads). *Let \mathcal{I} be an implementation. Let $e \in \llbracket \mathcal{I} \rrbracket$ be an execution (finite or infinite) of the form $e_1 \cdot (aid, pid, \text{read}(\ell)) \cdot e_2$, where $aid \in \mathbb{N}$, $pid \in \mathbb{P}$ and $\ell \in \mathbb{N}^*$. Then, $e_1 \cdot e_2 \in \llbracket \mathcal{I} \rrbracket$.*

Lemma 4 shows that, given an infinite sequence of increasing finite executions $e_1 \dots, e_n, \dots$ that satisfy a fairness condition, the *limit* execution (which is infinite) also belongs to $\llbracket \mathcal{I} \rrbracket$. The fairness condition states that each broadcast that appears in an execution e_i must have corresponding receive actions for each of the other sites $pid \in \mathbb{P}$ in some executions e_j .

Definition 12. *Given an infinite sequence of finite sequences $e_1 \dots, e_n, \dots$, such that for all $i \geq 1$, $e_i \sqsubset e_{i+1}$, the limit e^∞ of $e_1 \dots, e_n, \dots$ is the (unique) infinite sequence such that for all i , $e_i \sqsubset e^\infty$.*

Lemma 4 (Limit). *Let \mathcal{I} be an implementation. Let $e_1 \dots, e_n, \dots$ be an infinite sequence of finite executions, such that for all $i \geq 1$, $e_i \in \llbracket \mathcal{I} \rrbracket$, $e_i \sqsubset e_{i+1}$, and such that for all $i \geq 1$, for all broadcast actions in e_i , and for all $pid \in \mathbb{P}$, there exists $j \geq 1$ such that e_j contains a corresponding receive action.*

Then, the limit e^∞ of $e_1 \dots, e_n, \dots$ belongs to $\llbracket \mathcal{I} \rrbracket$.

We finally prove in Lemma 5 that, given any finite execution e , it is possible to add a query action that returns a list containing all the elements W appearing in some write action of e . The proof relies on extending e into an infinite execution e^∞ with an infinite number of queries. Our convergence assumption then ensures that only finitely many of those queries can ignore W (that is, return a list that does not contain all elements of W). This shows that there exists a query operation (actually, infinite many) in e^∞ that returns a list containing all elements of W . We can therefore take the finite prefix of e^∞ that ends with this query operation.

Lemma 5 (Convergence). *Let \mathcal{I} be a convergent implementation. Let $e \in \llbracket \mathcal{I} \rrbracket$ be a finite execution and $pid \in \mathbb{P}$. Let $W \subseteq \mathbb{N}$ be the set of elements appearing in an update action of e , i.e., $W = \{d \in \mathbb{N} \mid \exists (aid, pid, \text{write}(d)) \in e\}$.*

Then, e can be extended in an execution $e \cdot e' \cdot (aid, pid, \text{read}(\ell)) \in \llbracket \mathcal{I} \rrbracket$ where $\ell \in \mathbb{N}^$ contains every element of W , i.e., $W \subseteq \{d \in \mathbb{N} \mid d \in \ell\}$. Moreover, we can define such an extension e' that does not contain any query or update actions.*

Proof. We build an infinite sequence of finite executions e_1, \dots, e_n, \dots , where for every $i \geq 1$, $e_i \in \llbracket \mathcal{I} \rrbracket$. Moreover, we have $e_1 = e$ and for every $i \geq 1$, $e_i \sqsubseteq e_{i+1}$, and e_{i+1} is obtained from e_i as follows.

For every broadcast action $(aid_1, pid_1, \text{broadcast}(mid, msg))$ in e_i , and for every $pid_2 \in \mathbb{P}$, if there is no receive action $(_, pid_2, \text{receive}(mid, msg))$ in e_i , then

we add one when constructing e_{i+1} . Moreover, if the message handler specifies that a message msg' should be sent when msg is received, we add a new broadcast action that sends msg' , immediately following the receive action. Finally, using Lemma 2, we add a query action (**read**) on site pid .

Then, we define e^∞ to be the limit of e_1, \dots, e_n, \dots . By Lemma 4, we have $e^\infty \in \llbracket \mathcal{I} \rrbracket$. Since \mathcal{I} is convergent, we know that e^∞ is eventually consistent. This ensures that for every $d \in W$, out of the infinite number of queries that belong to e^∞ , only finitely many do not contain d .

Therefore, there exists $i \geq 1$ such that e_i ends with a query action that contains every element of W . By construction, e_i is of the form $e \cdot e'' \cdot (aid, pid, \mathbf{read}(\ell))$. Using Lemma 3, we remove every query action that appears in e'' , and obtain an execution of the form $e \cdot e' \cdot (aid, pid, \mathbf{read}(\ell))$ where $\ell \in \mathbb{N}^*$ contains every element of W , and where e' does not contain any query or update actions. \square

6.3 Nothing Is Stronger Than MPC in a Distributed Setting

We now proceed with the proof that no convergent implementation is strictly stronger than MPC. We start with an implementation \mathcal{I} that is strictly stronger than MPC and derive a contradiction.

More precisely, using the lemmas proved in Sect. 6.2, we prove that any trace of MPC belongs to $\text{tr}(\llbracket \mathcal{I} \rrbracket)$. First, we show in Lemma 6 that this holds for finite traces, by using an induction on the number of write operations in the trace. For each write operation w , we apply Lemma 5 in order to force the sites to take into account w .

Lemma 6. *Let \mathcal{I} be a convergent implementation such that $\mathcal{I} \prec \text{MPC}$, and let t be a finite trace of MPC. Then, there is a finite execution $e \in \llbracket \mathcal{I} \rrbracket$ such that $\text{tr}(e) = t$.*

Proof. Let $t = (t_r, W)$. We proceed by induction on the size of W , denoted n .

Case $n = 0$. In that case, the set W is empty. First, by definition of $\llbracket \mathcal{I} \rrbracket$, we have $\varepsilon \in \llbracket \mathcal{I} \rrbracket$ where ε is the empty execution. Then, for each read operation in t , and using Lemma 2, we add a read operation to the execution. We obtain an execution $e \in \llbracket \mathcal{I} \rrbracket$.

We then have to prove that $\text{tr}(e) = t$, meaning that all the read operations of e return the empty list, as in t . By our assumption that $\mathcal{I} \prec \text{MPC}$, we know that $\text{tr}(e) \in \text{MPC}$. By definition of MPC, and since e contains no write operation, the Consistency property of MPC ensures that all the read actions of e return the empty list. Therefore, we have $\text{tr}(e) = t$, which concludes our proof.

Case $n > 0$. We consider two subcases. (1) There exists a write $w \in W$ whose value does not appear in t_r . We consider the trace $t' = (t_r, W \setminus \{w\})$. By definition of MPC, t' belongs to MPC, and we deduce by induction hypothesis that there exists an execution $e' \in \llbracket \mathcal{I} \rrbracket$ such that $\text{tr}(e') = t'$. By Lemma 1, we extend e' in an execution $e \in \llbracket \mathcal{I} \rrbracket$ so that $\text{tr}(e) = t$, which is what we wanted to prove.

(2) All the writes of W appear in the reads of t_r . By the Consistency and Prefix properties of MPC, there exists a non-empty sequence $\ell \in \mathbb{N}^+$ of elements from W , such that all read actions return a prefix of ℓ , and there exist read actions that return the whole list ℓ .

Let $\ell = \ell' \cdot d$, where $d \in \mathbb{N}$ is the last element of ℓ . Let t' be the trace $(t'_r, W \setminus \{d\})$, such that t'_r is the trace t_r where every query action labelled by ℓ is replaced by a query action labelled by ℓ' , and implicitly, every query action labelled by any prefix of ℓ' is unchanged. Let R the set of the newly added query actions, and let $P \subseteq \mathbb{P}$ be the set of site identifiers that appear in an action of R .

By definition of MPC, we have $t' \in \text{MPC}$. By induction hypothesis, we deduce that there exists a finite execution $e' \in \llbracket \mathcal{I} \rrbracket$ such that $\text{tr}(e') = t'$.

Then, by Lemma 1, we add at the end of e' an update action (on some site $pid \in \mathbb{P}$ and with some fresh $aid \in \mathbb{N}$), which is of the form $(aid, pid, \text{write}(d))$, so we get an execution $e'' \in \llbracket \mathcal{I} \rrbracket$ such that $\text{tr}(e'') = (t'_r, W \setminus \{d\} \cup \{d\}) = (t'_r, W)$.

Using Lemma 5, we extend e'' in an execution e''' by adding queries to the sites in P , as many as were replaced by queries in R . Since $\mathcal{I} \prec \text{MPC}$, and since by Lemma 5, the answers to these queries must contain all the elements of ℓ , we conclude that the only possible answer for all these queries is the entire list ℓ .

Finally, we use Lemma 3 to remove the queries R from e''' , and we obtain an execution in $\llbracket \mathcal{I} \rrbracket$ whose trace is t . \square

We then extend Lemma 6 to infinite executions.

Theorem 1. *Let \mathcal{I} be a convergent implementation. Then, \mathcal{I} is not strictly stronger than MPC: $\mathcal{I} \not\prec \text{MPC}$.*

Proof. Assume that \mathcal{I} is strictly stronger than MPC i.e. $\mathcal{I} \prec \text{MPC}$. Our goal is to prove that $\text{MPC} \preceq \mathcal{I}$ therefore leading to a contradiction. In terms of traces, we want to prove that $\text{MPC} \subseteq \text{tr}(\llbracket \mathcal{I} \rrbracket)$.

Let $t = (t_r, W) \in \text{MPC}$. We need to show that $t \in \text{tr}(\llbracket \mathcal{I} \rrbracket)$.

Case where t is finite. Proven in Lemma 6.

Case where t is infinite. Let $t_r = (A, <, \text{label})$. We first order all the query actions in A as a sequence $aid_1, \dots, aid_n, \dots$ such that for every $i \geq 1$, $\text{label}(aid_i) \sqsubseteq \text{label}(aid_{i+1})$, and for every $i, j \geq 1$, $aid_i < aid_j$ (in the program order of t_r) implies $i < j$. Defining such a sequence is possible thanks to the Monotonicity property of MPC.

For each $i \geq 1$, we define a *finite* trace t_i that contains all query actions aid_j with $j \leq i$, and the subset W_i of W that contains all elements appearing in these query actions, i.e. $W_i = \{d \in W \mid d \in \text{label}(aid_i)\}$. Our goal is to construct an execution $e_i \in \llbracket \mathcal{I} \rrbracket$ such that $\text{tr}(e_i) = t_i$, and such that for all $i \geq 1$, $e_i \sqsubset e_{i+1}$. We then define e^∞ as the limit of e_1, \dots, e_n, \dots . By Lemma 4, we have $e^\infty \in \llbracket \mathcal{I} \rrbracket$. Since $\text{tr}(e^\infty) = t$, we deduce that $t \in \text{tr}(\llbracket \mathcal{I} \rrbracket)$, which concludes the proof.

We now explain how to construct e_i , for every $i \geq 1$, by induction on i . Let e_0 be the empty execution and $t_0 = \text{tr}(e_0)$. For $i \geq 0$, we define e_{i+1} by starting from e_i , and extending it as follows. By induction, we know that $\text{tr}(e_i) = t_i$, and want to extend it into an execution e_{i+1} such that $\text{tr}(e_{i+1}) = t_{i+1}$.

The next step of the proof is similar to the proof of Lemma 5. For every broadcast action $(aid_1, pid_1, \text{broadcast}(mid, msg))$ in e_i , and for every $pid_2 \in \mathbb{P}$, if there is no receive action $(-, pid_2, \text{receive}(mid, msg))$ in e_i , then we add one when constructing e_{i+1} . Moreover, if the message handler specifies that a message msg' should be sent when msg is received, we add a new broadcast action that sends msg' , immediately following the receive action.

Then, similarly to the construction in Lemma 6, we add update and query actions (using Lemmas 1, 2, and 5) in order to obtain an execution e_{i+1} such that $\text{tr}(e_{i+1}) = t_{i+1}$. \square

7 Comparison with Other Consistency Criteria

Relation between MPC and other consistency criteria. Consistency criteria are usually defined in terms of *full traces* that contain both the read and write operations in the program order (see e.g., [8]). The definition of trace we used in this paper (Definition 7, Sect. 3) puts the writes in an unordered set, unrelated to the read operations. This choice is justified in large-scale, open, implementations, such as blockchain protocols. Indeed, in these systems, any participant can perform a write operation (e.g., a blockchain transaction), and the origin of the write has no relevance for the protocol.

When considering full traces, MPC as a consistency criterion is strictly weaker than strong consistency. Indeed, MPC allows a trace where a read preceded by a write on the same site ignores that write.

As explained in the introduction, MPC is not comparable to causal consistency. MPC allows full traces that causal consistency forbids and vice versa. Therefore, our result stating that nothing stronger than MPC that can be implemented in a distributed setting does not contradict earlier results of [4, 10], which show that nothing stronger than variants of causal consistency can be implemented.

Relation with Other Criteria When Using our Notion of a Trace. When using our notion of a trace, MPC is strictly stronger than causal consistency. First, MPC is stronger than causal consistency because every trace of MPC can be produced by a causally consistent system. The main reason is that our notion of a trace does not capture any causality relation. Moreover, there are some traces that causal consistency produces and that do not belong to MPC, e.g. a trace where Site 1 has a `read[1, 2]` operation, Site 2 has a `read[2, 1]`, and where `write(1)` and `write(2)` are not causally related as they happen at the *same time* (this explains that MPC is *strictly* stronger than causal consistency).

Moreover, it is interesting to note that, for our notion of a trace, the traces allowed by MPC are exactly the traces allowed by strong consistency. This entails that, if the replicated data type is used by clients that only have the observability defined by our traces, then there is no need to implement strong consistency. In short, MPC and strong consistency are indistinguishable to these clients.

8 Conclusion

We have investigated the question of what is the strongest consistency criterion that can be implemented when replicating a data structure, in distributed systems under availability and partition-tolerance requirements. Earlier work had established the impossibility of implementing strong consistency in such a system model, but left open the question of the strongest criteria that *can* be implemented. In this paper we have focused on the *Monotonic Prefix Consistency* (MPC) criterion. We proposed an implementation of MPC and showed that no criterion stronger than MPC can be implemented.

It is worth noting that blockchain protocols, such as the Bitcoin protocol [11], implement MPC with high probability: the traces that the protocol produces are traces that belong to MPC with high probability. This was shown in [12, 13]. More precisely, the authors proved that the blockchains of two honest participants are compatible, in the sense that one should be a prefix of the other with high probability, when ignoring the last blocks¹. This property is called *consistency* in [12], and it corresponds to the Prefix property we give in Sect. 4. Moreover, it was shown [12, 13] that the blockchain of an honest participant only grows over time. This property is called *future-self consistency* in [12], and it corresponds to the Monotonicity property we give in Sect. 4.

In future work we plan to investigate how the strongest achievable consistency criterion depends on observability – that is, the information encoded in a trace – and study conditions for the (non)existence of a strongest consistency criterion. We are also interested in extending our result to other system models. Specifically, answering the question of what is the strongest consistency criterion that can be implemented in systems where the origin of updates do matter for the protocol. Also, the question whether MPC is the strongest implementable consistency criterion in a *probabilistic* setting, remains open.

A Proof of Feasibility of MPC

Proposition 1. *Let \mathcal{I} be the implementation of Fig. 2. Then $\mathcal{I} \preceq \text{MPC}$.*

Proof. Let $e \in \llbracket \mathcal{I} \rrbracket$, we establish an inductive invariant that holds for every finite prefix e' of e . Let A be the set of action identifiers of e' . Let $\ell \in \mathbb{N}^*$ be the sequence of values that appear in a broadcast message **Apply** from Site 1, in the order they appear in e' .

Let \mathbb{P} be the set of process identifiers. For each site $pid \in \mathbb{P}$, consider the unique run r for the projection $e'|_{pid}$, and let $\ell_{pid} \in \mathbb{N}^*$ be the sequence maintained in the local state of Site pid at the end of the run r .

Let $t' = \text{tr}(e')$ be the trace of e' , with $t' = (t_r, W)$.

Then, we have the following properties.

¹ In Bitcoin-like protocols, the most recent blocks are ignored as they are considered unsafe to use until newer blocks are appended after them.

1. For every **Apply**(d) message with $d \in \mathbb{N}$ that appears in e' (from Site 1), we have $d \in W$.
2. For every **Forwarded**(d) message with $d \in \mathbb{N}$ that appears in e' (from Site i with $i > 1$), we have $d \in W$.
3. The elements of ℓ are in W .
4. For every $pid \in \mathbb{P}$, $l_{pid} \sqsubseteq \ell$.
5. For every query $(-, pid, \text{read } \ell')$ in e with $pid \in \mathbb{P}$, we have $\ell' \sqsubseteq \ell_{pid}$.
6. **Consistency:** For all $aid \in A$, and for any element $d \in \mathbb{N}$ of $\text{label}(aid)$, we have $d \in W$.
7. **Prefix:** For any all $aid, aid' \in A$, $\text{label}(aid) \sqsubseteq \text{label}(aid')$ or $\text{label}(aid') \sqsubseteq \text{label}(aid)$.
8. **Monotonicity:** For all $aid, aid' \in A$, if $aid < aid'$, then $\text{label}(aid) \sqsubseteq \text{label}(aid')$.

We can see that this invariant holds for the empty execution, and that any action that the implementation can take maintains it.

B Closure Properties of Implementations

Lemma 2 (Query Availability). *Let \mathcal{I} be an implementation. Let $e \in \llbracket \mathcal{I} \rrbracket$ be a finite execution and $pid \in \mathbb{P}$. Then, there exist $aid \in \mathbb{N}$ and $\ell \in \mathbb{N}^*$ such that the execution $e' = e \cdot (aid, pid, \text{read}(\ell))$ belongs to $\llbracket \mathcal{I} \rrbracket$.*

Proof. Similar to the proof of Lemma 1, but using the query handler, instead of the update handler. This proof is also simpler, as there is no need to consider messages, since the query handler cannot broadcast any message. Therefore, in this proof, only case 1 needs to be considered. \square

Lemma 3 (Invisible Reads). *Let \mathcal{I} be an implementation. Let $e \in \llbracket \mathcal{I} \rrbracket$ be an execution (finite or infinite) of the form $e_1 \cdot (aid, pid, \text{read}(\ell)) \cdot e_2$, where $aid \in \mathbb{N}$, $pid \in \mathbb{P}$ and $\ell \in \mathbb{N}^*$. Then, $e_1 \cdot e_2 \in \llbracket \mathcal{I} \rrbracket$.*

Proof. This is a direct consequence of Definition 5, which specifies that query actions do not modify the local state of sites, and do not broadcast messages. \square

Lemma 4 (Limit). *Let \mathcal{I} be an implementation. Let e_1, \dots, e_n, \dots be an infinite sequence of finite executions, such that for all $i \geq 1$, $e_i \in \llbracket \mathcal{I} \rrbracket$, $e_i \sqsubset e_{i+1}$, and such that for all $i \geq 1$, for all broadcast actions in e_i , and for all $pid \in \mathbb{P}$, there exists $j \geq 1$ such that e_j contains a corresponding receive action.*

Then, the limit e^∞ of e_1, \dots, e_n, \dots belongs to $\llbracket \mathcal{I} \rrbracket$.

Proof. According to Definition 6, we have three points to prove.

- (1) (Projection) First, we want to show that, for all $pid \in \mathbb{P}$, the projection $e^\infty|_{pid}$ follows \mathcal{I} . For all $i \geq 1$, we know that $e_i \in \llbracket \mathcal{I} \rrbracket$, and deduce that $e_i|_{pid}$ follows \mathcal{I} . Let r_i be the run of $e_i|_{pid}$. Note that for all $i \geq 1$, we have $r_i \sqsubset r_{i+1}$. Let r_{pid}^∞ be the limit of the runs r_1, \dots, r_n, \dots . By construction, r_{pid}^∞ is a run of $e^\infty|_{pid}$, which shows that $e^\infty|_{pid}$ follows \mathcal{I} .

- (2) (Causality) We need to prove that every receive action σ in e^∞ has a corresponding broadcast action σ' that precedes it in e^∞ . Let e_i be a prefix of e^∞ that contains σ . Since $e_i \in \llbracket \mathcal{I} \rrbracket$, we know that there exists a broadcast action σ' corresponding to σ , and that precedes σ in e_i . Finally, since $e_i \sqsubset e^\infty$, σ' precedes σ in e^∞ .
- (3) (Fairness) We want to prove that for every broadcast action σ of e^∞ and for every site $pid \in \mathbb{P}$, there exists a corresponding receive action σ' . Let e_i be a prefix of e^∞ that contains σ . By assumption of the current lemma, there exists $j \geq 1$ such that e_j contains a receive action σ' corresponding to σ . Moreover, since $e_j \sqsubset e^\infty$, σ' belongs to e^∞ , which concludes our proof. \square

References

1. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* **12**(3), 463–492 (1990)
2. Brewer, E.: CAP twelve years later: how the “Rules” have changed. *Computer* **45**(2), 23–29 (2012)
3. Gilbert, S., Lynch, N.A.: Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* **33**(2), 51–59 (2002)
4. Attiya, H., Ellen, F., Morrison, A.: Limitations of highly-available eventually-consistent data stores. *IEEE Trans. Parallel Distrib. Syst.* **28**(1), 141–155 (2017)
5. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7), 558–565 (1978)
6. Terry, D.: Replicated data consistency explained through baseball. Technical report MSR-TR-2011-137, Microsoft Research, October 2011
7. Guerraoui, R., Pavlovic, M., Seredinschi, D.A.: Trade-offs in replicated systems. *IEEE Data Eng. Bull.* **39**, 14–26 (2016)
8. Burckhardt, S.: Principles of Eventual Consistency. Now Publishers, October 2014
9. Bouajjani, A., Enea, C., Hamza, J.: Verifying eventual consistency of optimistic replication systems. In: Jagannathan, S., Sewell, P. (eds.) *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20–21 January 2014*, pp. 285–296. ACM (2014)
10. Mahajan, P., Alvisi, L., Dahlin, M.: Consistency, availability, convergence. Technical report TR-11-22, Computer Science Department, University of Texas at Austin, May 2011
11. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
12. Pass, R., Seeman, L., Shelat, A.: Analysis of the blockchain protocol in asynchronous networks. In: Coron, J.-S., Nielsen, J.B. (eds.) *EUROCRYPT 2017*. LNCS, vol. 10211, pp. 643–673. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56614-6_22
13. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: analysis and applications. In: Oswald, E., Fischlin, M. (eds.) *EUROCRYPT 2015*. LNCS, vol. 9057, pp. 281–310. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_10



A Modest Security Analysis of Cyber-Physical Systems: A Case Study

Ruggero Lanotte¹, Massimo Merro²(✉), and Andrei Munteanu²

¹ Dipartimento di Scienza e Alta Tecnologia, Università dell'Insubria, Como, Italy
ruggero.lanotte@uninsubria.it

² Dipartimento di Informatica, Università degli Studi di Verona, Verona, Italy
{massimo.merro, andrei.munteanu}@univr.it

Abstract. *Cyber-Physical Systems* (CPSs) are integrations of networking and distributed computing systems with physical processes. Although the range of applications of CPSs include several critical domains, their verification and validation often relies on simulation-test systems rather than formal methodologies. In this paper, we use a recent version of the expressive MODEST TOOLSET to implement a non-trivial engineering application, and test its *safety model checker* `prohver` as a formal instrument to statically detect a variety of *cyber-physical attacks*, *i.e.*, attacks targeting *sensors* and/or *actuators*, with potential physical consequences. We then compare the effectiveness of the MODEST TOOLSET and its safety model checker in verifying CPS security properties when compared to other state-of-the-art model checkers.

1 Introduction

Cyber-Physical Systems (CPSs) are integrations of networking and distributed computing systems with physical processes, where feedback loops allow the latter to affect the computations of the former and vice versa. CPSs have three main components: the *physical plant*, *i.e.*, the physical process that is managed by the CPS; the *logics*, *i.e.*, controllers, intrusion detection systems (IDSs), and supervisors that govern and control the physical process; the connecting *network*.

Historically, CPSs relied on proprietary technologies and were implemented as stand-alone networks in physically protected locations. However, in recent years the situation has changed considerably: commodity hardware, software and communication technologies are used to enhance the connectivity of these systems and improve their operation.

This evolution has triggered a dramatic increase in the number of attacks to the security of cyber-physical and critical systems, *e.g.*, manipulating sensor readings and, in general, influencing physical processes to bring the system into a state desired by the attacker. Some notorious examples are: (i) the *Stuxnet* worm, which reprogrammed PLCs of nuclear centrifuges in Iran [7], (ii) the attack on a sewage treatment facility in Queensland, Australia, which manipulated

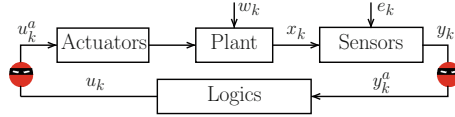


Fig. 1. A threat model for CPSs

the SCADA system to release raw sewage into local rivers [27], or the (iii) the recent cyber-attack on the Ukrainian power grid, again through the SCADA system [15].

The common feature of the systems above is that they are all safety critical and failures may cause catastrophic consequences. Thus, the concern for consequences at the physical level puts *CPS security* apart from standard *IT security*, and demands for *ad hoc* solutions to properly address such novel research challenges.

The *physical plant* of a CPS is often represented by means of a *discrete-time state-space model*¹ consisting of two *difference equations* of the form

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k + w_k \\ y_k &= Cx_k + e_k \end{aligned}$$

where $x_k \in \mathbb{R}^n$ is the current (*physical*) *state*, $u_k \in \mathbb{R}^m$ is the *input* (*i.e.*, the control actions implemented through actuators), $w_k \in \mathbb{R}^n$ is the *system uncertainty*, $y_k \in \mathbb{R}^p$ is the *output* (*i.e.*, the measurements from the sensors), and $e_k \in \mathbb{R}^p$ is the *measurement error*. A , B , and C are matrices modelling the dynamics of the physical system.

Cyber-physical attacks typically tamper with both the physical (sensors and actuators) and the cyber layer. In particular, cyber-physical attacks may affect directly the sensor measurements or the controller commands (see Fig. 1):

- *Attacks on sensors* consist of reading and possibly replacing the genuine sensor measurements y_k with fake measurements y_k^a .
- *Attacks on actuators* consist of reading, dropping and possibly replacing the genuine controller commands u_k with malicious commands u_k^a , affecting directly the actions the actuators may execute.

One of the central problem in the *safety verification* of CPSs is the *reachability* problem: can an unsafe state be reached by an execution of the system (possibly under attack) starting from a given initial state? In general, the reachability problem for *hybrid systems* (and hence CPSs) is stubbornly undecidable, although boundaries of decidability have been extensively explored in the past couple of decades [1, 14, 17, 26, 30]. Thus, despite the undecidability of the safety problem, a number of formal verification tools for hybrid systems have been recently proposed, based on approximation techniques to obtain an estimation of the set of reachable states: SpaceEx [10], PHAVer [9] and SpaceEx AGAR [3], for

¹ See [31] for a taxonomy of the time-scale models used to represent CPSs.

linear/affine dynamics, and `HSolver` [23], `C2E2` [6] and `FLOW*` [5], for non-linear dynamics. Among these, the hybrid solver `PHAVer` addresses the *exact verification* of safety properties of hybrid systems with piecewise constant bounds on the derivatives, so-called *rectangular hybrid automata* [14]. *Affine dynamics* are handled by *on-the-fly overapproximation* and partitioning of the state space based on user-provided constraints and the dynamics of the system. To force termination and manage the complexity of the computations, methods to conservatively limit the number of bits and constraints are adopted.

Contribution. We implement in the `MODEST TOOLSET` [12], an integrated collection of tools for the design and the formal analysis of *stochastic hybrid automata*, a simple but totally realistic and nuanced cyber-physical system. The example has been proposed by Lanotte et al. [19] to highlight different classes of attacks on sensors and actuators, in a way that is basically independent on the size of the system. Our case study is implemented in the main modelling language `HMODEST` [11], a process-algebra based language that has an expressive programming language-like syntax to design complex systems.

The current version of the toolset comprises several analysis backends, in particular it provides a *safety model checker*, called `prohver`, that relies on a modified version of the hybrid solver `PHAVer` [9]. We use `prohver` to analyse three simple but significant cyber-physical attacks targeting sensors and/or actuators of our case study by compromising either the corresponding physical device or the communication network used by the device. The three attacks have already been carefully studied in [19] focussing on the time aspects of the attacks (begin, duration, *etc.*) and the physical impact on the system under attack (deadlock, unsafe behaviour, *etc.*). Here, we test the safety model checker `prohver` as an automatic tool to get the same (or part of the) results that have been manually proved in [19]. We then compare its effectiveness in verifying CPS security properties, when compared to other state-of-the-art models checkers, such as `PRISM` [16], `UPPAAL` [2] and `Real-Time Maude` [22].

Outline. In Sect. 2 we give a brief description of the `MODEST TOOLSET`. In Sects. 3 and 4 we first describe and then implement in `HMODEST` our case study. In Sect. 5 we put under stress the safety model checker `prohver` for a security analysis of our case study under three different cyber-physical attacks. In Sect. 6 we draw conclusions, compare the expressivity of the `MODEST TOOLSET` with respect to other model-checkers, and discuss related work in the context of formal methods for CPS security.

2 The `MODEST TOOLSET`

The `MODEST TOOLSET` [4] has been originally proposed as an integrated collection of tools for the design and the formal analysis of stochastic timed automata (STA). More recently, it has been extended to add differential equations and inclusions as an expressive way to model continuous system evolutions [11]. Thus, the current version of the toolset [12] is now based on the rich semantic foundation of networks of stochastic hybrid automata (SHA), *i.e.*, sets of

automata that run asynchronously and can communicate via shared actions and global variables.

SHA combine three key modelling concepts:

- *Continuous dynamics* to represent continuous processes, such as physical laws or chemical reactions, the evolution of general continuous variables over time can be described using differential (in)equations.
- *Nondeterminism* to model concurrency (via an interleaving semantics) or the absence of knowledge over some choice, to abstract from details, or to represent the influence of an unknown environment.
- *Probability* to model situations in which an outcome is uncertain but the probabilities of the outcomes are known; these choices may be discrete (“probabilistic”) or continuous (“stochastic”).

The current version of the MODEST TOOLSET comprises analysis backends for model checking timed automata (`mctau`) and probabilistic timed automata (`mcpta`), and for statistical model checking of stochastic timed automata (`modes`). However, in this paper we focus on the *safety model checker* for SHA, called `prohver`, that relies on a modified version of the hybrid solver `PHAVer` [9].

The main modelling language is HMODEST [11], a process-algebra based language that has an expressive programming language-like syntax to design complex models in a reasonably concise manner. Here, we provide a brief and intuitive explanation of the main constructs.

A HMODEST specification consists of a sequence of declarations (constants, variables, actions, and sub-processes) and a main process behaviour. The most simple process behaviour is expressed by (prefixing) *actions* that may be used for synchronising parallel components. The construct *do* serves to model loops, *i.e.*, unguarded iterations that can be exited via the special action *break*. There is a construct *par* to launch two or more processes in *parallel*, according to an interleaving semantics. The construct *alt* models *nondeterministic choice*. The *invariant* construct serves to control the evolution of continuous variables. Furthermore, all constructs can be decorated with guards, to represent enabling conditions, by means of the *when* construct. We can use both *invariant* and *when* constructs to specify that a behaviour should be executed after a precise amount of time. Thus, we can write *invariant*($c \leq k$) *when* ($c \geq k$) $P()$, where c is a clock variable and k a real value, to model that the process $P()$ may start its execution only after k time units; if $k = 0$ then the execution of $P()$ may start immediately.

In order to better explain these constructs, we model a small example described by means of a standard timed process-calculus notation (say, Hennessy and Regan’s TPL [13]). Consider a *Master* and a *Slave* process that may synchronise via a private synchronisation channel *sync*, and use a private channel *ins* to allow the *Master* to send instructions to the *Slave*. Depending on the received instructions, the *Slave* either synchronises with the *Master* and then restarts, or sleeps for one time unit and then ends its execution. Once synchronised, the *Master* sleeps for two time units. Formally,

$$\begin{aligned}
Master &\stackrel{\text{def}}{=} ins(\text{go}); sync; sleep(2); ins(\text{end}) \\
Slave &\stackrel{\text{def}}{=} ins(i); \text{if } (i = \text{go}) \{ sync; Slave \} \text{ else } \{ sleep.\text{stop} \}
\end{aligned}$$

and the compound system is given by

$$(Master \parallel Slave) \setminus \{ins, sync\}.$$

Figure 2 shows an implementation in HMODEST of the system above. Both master and slave declare private clocks that are reset each time is necessary to impose a specific time delay. Value-passing communication is implemented via the two actions *go* and *end*; the testing via nondeterministic choice.

```

1 // declarations
2 action sync, go, end;
3 process Master(){ // process declaration
4   clock cm;
5   invariant (cm <= 0) when(cm >= 0) go; sync {= cm = 0 =};
6   invariant (cm <= 2) when(cm >= 2) end
7 }
8 process Slave(){ // process declaration
9   clock cs;
10  do{ alt{ :: go; sync
11        :: end {= cs = 0 =}; invariant(cs <= 1) when(cs >= 1) break
12      }
13 }
14 }
15
16 // main behaviour
17 par { :: Master() :: Slave()
18 }

```

Fig. 2. Master and Slave processes in HMODEST

Besides these operators, the case study that we will present in the next section includes specifications over continuous variables, such as constraints over the derivate of continuous variables of the form $a \leq \dot{x} \leq b$, with a and b constant (as in *rectangular hybrid automata*), or nondeterministic initialisations of the form $z \in [a, b]$. The former requirement is realised in HMODEST by means of an invariant construct: $invariant(der(x \geq a) \ \&\& \ der(x \leq b))$. The latter constraint is implemented via the *any* construct. For instance, $any(z, z \geq a \ \&\& \ z \leq b)$ returns a value nondeterministically chosen in the real interval $[a, b]$.

The safety model checker *prohver* allows the verification of reachability properties of the form $Pmax(\Diamond_{time \leq T} e)$. This query returns an upper bound of the probability of reaching the states characterised by the deterministic expression e within the time bound T .² Moreover, as the models may be nondeterministic, $Pmax()$ computes the probability over all possible resolutions of nondeterminism.

² Later in the paper, we will show how to get the exact probability.

3 A Case Study

In this section, we describe the case study recently introduced in [19]. Here, we wish to remark that while the example is quite simple, it is actually far from trivial and designed to describe a wide number of attacks. Furthermore, for simplicity, in the description of the case study we use a discrete-time model, although in its implementation we will adopt a continuous notion of time.

Consider a CPS *Sys* in which the temperature of an engine is maintained within a certain range by means of a cooling system controlled by a controller. The system is also equipped with a IDS that does runtime safety verification. Let's describe both the physical and the cyber component of the CPS *Sys*.

The physical environment of *Sys* is constituted by:

- a *variable temp*, initialised to 0, for the current temperature of the engine;
- a *sensor sens* measuring the temperature of the engine;
- an *actuator cool* to turn on/off the cooling system; *cool* ranges over the set $\{-1, +1\}$ to denote active and inactive cooling, respectively;
- the *evolution equation* $temp_{k+1} = temp_k + cool_k + w_k$, where $w_k \in [-0.4, +0.4]$ denotes the *uncertainty* associated to *temp*; thus the variable *temp* is increased (*resp.*, is decreased) of one degree per time unit if the cooling system is inactive (*resp.*, active) up to a bounded uncertainty w_k ;
- a *measurement equation* $sens_k = temp_k + e_k$, where $e_k \in [-0.1, +0.1]$ denotes the *noise* associated to the sensor *sens*;
- an *invariant function* returning the Boolean true if the state variable *temp* lays in the interval $[0, 20]$, false otherwise;
- a *safety function* returning the Boolean true if the safety conditions are satisfied, false otherwise; the safety of the CPS depends on a (fictitious) variable *stress* keeping track of the level of stress of the mechanical parts of the engine due to high temperatures; *stress* ranges over the set $\{0, 1, 2, 3, 4, 5\}$, where 0 means no stress and 5 high stress; formally, $stress_{k+1} = \min(5, stress_k + 1)$ if $temp_k > 9.9$, while $stress_{k+1} = 0$ if $temp_k \leq 9.9$.

Let us define the cyber component of the CPS *Sys*. For simplicity, we use a simple process-calculus notation similar to that of Lanotte and Merro's CaIT [18]. The logics of *Sys* is modelled by means of two parallel processes: *Ctrl* and *IDS*. The former models the *controller* activity, consisting in reading the temperature of the engine and in governing the cooling system; whereas the latter models a simple *intrusion detection system* that attempts to detect and signal abnormal behaviours of the system. Intuitively, *Ctrl* senses the temperature of the engine via the sensor *sens* (reads the sensor) at each time slot. When the *sensed temperature* is above 10° , the controller activates the coolant via the actuator *cool* (sending a command to the actuator). The cooling activity is maintained for 5 consecutive time units. After that time, the controller synchronises with the *IDS* component via a synchronisation channel *sync*, and then waits for *instructions*, via a value-passing channel *ins*. The *IDS* component checks whether the *sensed temperature* is still above 10. If this is the case, it sends an *alarm* of "high temperature", via a specific channel, and then says to *Ctrl* to keep cooling for

a further 5 time units; otherwise, if the temperature is not above 10, the *IDS* component requires *Ctrl* to stop the cooling activity. More formally,

```

Ctrl = read sens(x).if (x > 10) {Cooling} else {sleep.Ctrl}
Cooling = write cool(on).sleep(5).Check
Check = sync.ins(y).if (y = keep_cooling) {sleep(5).Check}
        else {write cool(off).sleep.Ctrl}
IDS = sync.read sens(x).if (x > 10) ins(keep_cooling).{alarm(high_temp).sleep.IDS}
        else {ins(stop).sleep.IDS}

```

The whole cyber component of *Sys* is given by the parallel composition of the two processes *Ctrl* and *IDS* in which the channels *sync* and *ins* have been restricted: $(Ctrl \parallel IDS) \setminus \{sync, ins\}$.

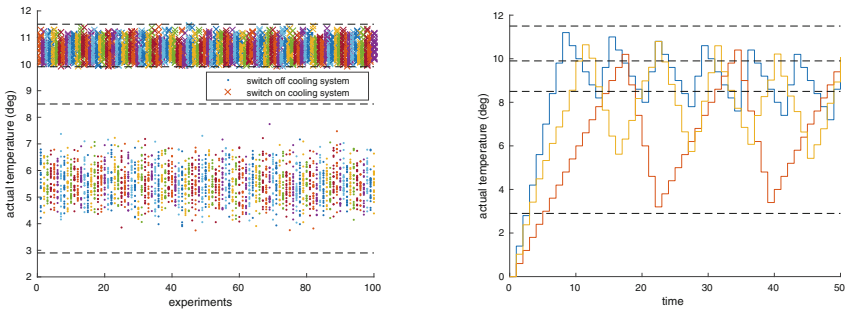


Fig. 3. Simulations in MATLAB of *Sys* (Color figure online)

In Fig. 3, the left graphic collect a campaign of 100 simulations of our engine in MATLAB, lasting 250 time units each, showing that the value of the state variable *temp* when the cooling system is turned on (*resp.*, off) lays in the interval $(9.9, 11.5]$ (*resp.*, $(2.9, 8.5]$); these bounds are represented by the dashed horizontal lines. The right graphic of the same figure shows three possible evolutions in time of the state variable *temp*: (i) the first one (in red), in which the temperature of the engine always grows as slow as possible and decreases as fast as possible; (ii) the second one (in blue), in which the temperature always grows as fast as possible and decreases as slow as possible; (iii) and a third one (in yellow), in which, depending whether the cooling is off or on, temperature grows or decreases of an arbitrary offset laying in the interval $[0.6, 1.4]$.

4 An Implementation in HMODEST

In this section, we provide our implementation in HMODEST of the case study presented in the previous section. The whole system is divided in three high level processes running in parallel (see Fig. 4):

```

1 // global clock and global action declarations
2 clock global_clock;
3 action on, off;
4 ...
5 // global variable declarations
6 var sens = 0; der(sens) = 0;
7 bool safe = true;
8 bool is_deadlock = false;
9 ...
10 //process declarations
11 process Plant() {
12   var temp = 0;
13   ...
14   par { :: Engine() :: Sensors() :: Actuators() :: Safety() }
15 }
16 process Logics() {
17   ...
18   par { :: Ctrl() :: IDS() }
19 }
20 process Network() {
21   ...
22   par { :: Proxy_actuator() :: Proxy_sensor() }
23 }
24
25 // main
26 par { :: Plant() :: Logics() :: Network() }

```

Fig. 4. Implementation in HMODEST of *Sys*

- `Plant()`, modelling the *physical aspects* of the system;
- `Logics()`, describing the *logical (or cyber) component* of a CPS;
- `Network()`, representing the *network* connecting `Plant()` and `Logics()`.

The process `Plant()` consists of the parallel composition of four processes: `Engine()`, `Actuators()`, `Sensors()` and `Safety()` (see Fig. 5). The former models the dynamics of the variable *temp* depending on the cooling activity. The temperature evolves in a continuous manner, and its rate is described by means of differential inclusions of the form $a \leq \dot{x} \leq b$ implemented via the construct *invariant*. The *on* action triggers the coolant and drives the process `Engine()` into a state `CoolOn()` in which the temperature decreases at a rate comprised in the range $[-DT-UNCERT, -DT+UNCERT]$. On the other hand, in the presence of a *off* action the engine moves into a `CoolOff()` state in which the coolant is turned off, so that the temperature increases at a rate ranging in $[DT-UNCERT, DT+UNCERT]$.

The second parallel component of `Plant()` is the process `Sensors()` that receives the requests to read the temperature, originating from the `Logics()`, and serves them according to the measurement equation seen in the previous section. This is modelled by updating the variable *sens* with an arbitrary real value laying in the interval $[temp - NOISE, temp + NOISE]$.

The process `Actuators()` relays the commands of the controller `Ctrl()` to the `Engine()` to turn on/off the cooling system.

The last parallel component of the process `Plant()` is the process `Safety()`. This process defines a local variable *stress* depending on the temperature reached by the engine; we recall that $stress = 0$ denotes no stress while $stress = 5$ represents maximum stress. Here, is worth mentioning that the variable *stress*

```

1  const real DT = 1;
2  const real UNCERT = 0.4; // uncertainty of variable temp
3  const real NOISE = 0.1; // sensor noise
4  clock c;
5
6  process Engine() {
7    process CoolOn() {
8      invariant( der(temp) >= (-DT - UNCERT) && der(temp) <= (-DT + UNCERT) )
9      alt { :: on; CoolOn() :: off; CoolOff() }
10   }
11   process CoolOff() {
12     invariant( der(temp) >= (DT - UNCERT) && der(temp) <= (DT + UNCERT) )
13     alt { :: on; CoolOn() :: off; CoolOff() }
14   }
15   CoolOff()
16 }
17
18 process Sensors() {
19   do { // detect temperature and write it in variable sens
20     read_sensor {= sens = any(z, z >= temp - NOISE && z <= temp + NOISE), c = 0 =};
21     invariant(c <= 0) when(c >= 0) ack_sensor
22   }
23 }
24
25 process Actuators(){
26   do { :: cool_on_actuator {= c = 0 =}; invariant(c <= 0) when(c >= 0) on // cool on
27       :: cool_off_actuator {= c = 0 =}; invariant(c <= 0) when(c >= 0) off // cool off
28   }
29 }
30
31 process Safety(){
32   var stress = 0; der(stress) = 0; // no continuous dynamics for stress
33   do { invariant(c <= 0) when(c >= 0)
34     alt { :: when(temp >= 0 && temp <= 20) // invariant is preserved
35           alt { :: when(temp > 9.9 && stress <= 3) {= stress = stress+1 =}
36                 :: when(temp <= 9.9) {= stress = 0, safe = true =}
37                 :: when(temp > 9.9 && stress >= 4) {= stress = 5, safe = false =}
38               // safety is violated
39           }
40     :: when(temp > 20 || temp < 0) {= is_deadlock = true =}; stop // system deadlock
41   };
42   invariant(c <= 1) when(c >= 1) {= c = 0 =} // move to the next time unit
43 }
44 }

```

Fig. 5. Plant() sub-processes

could be implemented either as a bounded integer variable, which would increase the discrete complexity of the underlying hybrid automaton, or as a continuous variable with dynamics set to zero (*i.e.*, $der(stress) = 0$) that would increase the continuous complexity of the automaton. We have adopted the second option as it ensures better performances. The Safety() process sets the global Boolean variable *safe* to false only when the system reaches the maximum stress, *i.e.*, $stress = 5$, and reset it to true otherwise. Thus, this variable says when the CPS is currently in a state that is violating the *safety conditions*. Similarly, the global Boolean variable *is_deadlock* is set to true whenever the system invariant is violated; in that case the whole CPS stops.

The process Logics() consists of the parallel composition of two processes: Ctrl() and IDS() (see Fig. 6). The former senses the temperature by triggering a *read_sensor_ctrl* action to request a measurement and waits for an

```

1  clock c;
2  process Ctrl() {
3    process Check() {
4      do{ invariant(c <= 0) when(c >= 0) tau;
5          invariant(c <= 5) when(c >= 5) {= c = 0 =}; // keep cooling for 5 time units
6          invariant(c <= 0) when(c >= 0) sync_ids; // activate IDS
7          alt { // wait for instructions
8            :: keep_cooling {= c = 0 =} // keep cooling a further 5 time units
9            :: stop_cooling {= c = 0 =};
10         invariant(c <= 0) when(c >= 0) set_cool_off; // turn off the coolant
11         invariant(c <= 1) when(c >= 1) {= c = 0 =}; // move to the next time slot
12         invariant(c <= 0) when(c >= 0) break // returns the control to Ctrl()
13       }
14     }
15   }
16   // main Ctrl()
17   do { invariant(c <= 0) when(c >= 0) read_sensor_ctrl; // request temperature sensing
18       ack_sensor_ctrl {= c = 0 =};
19       invariant(c <= 0) when(c >= 0)
20       alt { :: when(sens <= 10) tau {= c = 0 =}; // temperature is ok
21           invariant(c <= 1) when(c >= 1) {= c = 0 =} // move to the next time slot
22           :: when(sens > 10) set_cool_on {= c = 0 =}; // turn on the cooling
23           invariant(c <= 0) when(c >= 0) Check() // check whether temperature drops
24         }
25     }
26 }
27
28 process IDS() {
29   do{ sync_ids {= c = 0 =};
30       invariant(c <= 0) when(c >= 0) read_sensor_ids; // request temperature sensing
31       ack_sensor_ids;
32       invariant(c <= 0) when(c >= 0)
33       alt { :: when(sens <= 10) stop_cooling // temperature is ok
34           :: when(sens > 10) keep_cooling; // temperature is not ok, keep cooling
35           invariant(c <= 0) when(c >= 0) {= alarm = true =}; // fire the alarm
36           invariant(c <= 0) when(c >= 0) {= alarm = false =}
37         }
38     }
39 }

```

Fig. 6. Logics() sub-processes

ack_sensor_ctrl action to read the measurement in the variable *sens*. Depending on the value of *sens* the controller decides whether to activate or not the cooling system. If $sens \leq 10$ the process sleeps for one time unit and then check the temperature again. If $sens > 10$ then the controller activates the coolant by emitting the *set_cool_on* action that will reach the Engine() (via the Network()'s proxy). Afterwards the control passes to the process Check() that verifies whether the current cooling activity is effective in dropping the temperature below 10. The process Check() maintains the cooling activity for 5 consecutive time units. After that, it synchronises with the process IDS() via the action *sync_ids*, and waits for instructions from IDS(): (i) *keep cooling* for other 5 time units and then check again, or (ii) *stop the cooling* activity and returns. These two instructions are represented by means of the actions *keep_cooling* and *stop_cooling*, respectively.

The second component of the process Logics() is the process IDS(). The IDS() process waits for the synchronisation action *sync_ids* from Check(). Then, it triggers the action *read_sensor_ids* to request a measurement and waits for the *ack_sensor_ids* action to read the measurement. If $sens \leq 10$ it signals to

Ctrl() to stop cooling (via the action), otherwise, if $sens > 10$, it signals to keep cooling and fires an *alarm* by setting a global Boolean variable *alarm* to true (for verification reasons we immediately reset this variable to false).

The process Network() consists of the parallel composition of two processes: Proxy_actuator() and Proxy_sensor() (see Fig. 7). The former provides the remote actuation. Basically, it forwards the actuators commands originating from the process Ctrl() to the process Actuators(). The process Proxy_sensor() waits for requests of measurement originating from processes Ctrl() or IDS() (we use different actions for each of them) and relay these requests to the process Sensor() that implements the measurement equation. When the temperature has been detected an ack signal is returned and propagated up to the requesting process.

```

1  process Network() {
2    clock c;
3    process Proxy_actuator() {
4      do { alt { :: set_cool_on {= c = 0 =};
5                invariant(c <= 0) when(c >= 0) cool_on_actuator
6                :: set_cool_off {= c = 0 =};
7                invariant(c <= 0) when(c >= 0) cool_off_actuator
8              }
9      }
10   }
11  process Proxy_sensor(){
12    do { alt { :: read_sensor_ctrl {= c = 0 =};
13              invariant(c <= 0) when(c >= 0) read_sensor;
14              ack_sensor;
15              invariant(c <= 0) when(c >= 0) ack_sensor_ctrl
16              :: read_sensor_ids {= c = 0 =};
17              invariant(c <= 0) when(c >= 0) read_sensor;
18              ack_sensor;
19              invariant(c <= 0) when(c >= 0) ack_sensor_ids
20            }
21    }
22  }
23
24  par{ :: Proxy_actuator() :: Proxy_sensor() }
25 }

```

Fig. 7. Network() process

Verification. We conduct our safety verification using 4 notebooks with the following set-up: (i) 2.8 GHz Intel i7 7700 HQ, with 16 GB memory, and Linux Ubuntu 16.04 operating system; (ii) MODEST TOOLSET Build 3.0.23 (2018-01-19).

In order to assess the correct functioning of our implementation, we verify a number of properties of our CPS *Sys* by means of the safety model checker **prohver**. Here, it is important to recall that **prohver** relies on the hybrid solver **PHAVer** which computes an *overapproximation* of the reachable states to ensure termination and accelerate convergence [9]. As consequence, the probability returned by the verification of a generic property $Pmax(\diamond_{Te_{prop}})$ is an upper bound of the exact probability, and hence it is significant only when equal to zero (*i.e.*, when the property is not satisfied). However, as our CPS *Sys* presents a *linear dynamics* it is possible to compute the exact probability by launching our analyses

with the `NO_CHEAP_CONTAIN_RETURN_OTHERS` flag (see [8]) which enables the exact computation of the reachable sets, with obvious implications on the time required to complete the analyses. As our case study does not present a probabilistic behaviour, the results of our analyses will always range in the set $\{0, 1\}$ (unsatisfied/satisfied) with a 100% accuracy.

Furthermore, as a formula $\Box e$ is satisfied if and only if $\Diamond \neg e$ is unsatisfied, we can use `prohver` to verify properties expressed in terms of *time bounded LTL* formulae of the form $\Box_{[0,T]} e_{prop}$ or $\Diamond_{[0,T]} e_{prop}$. Actually, in our analyses we will always verify properties of the form $\Box_{[0,T]} e_{prop}$, relying on the quicker overapproximation when proving that the property is satisfied, and resorting to the slower exact computation when proving that the property is not satisfied.

Thus, we have formally proved that in all possible executions that are (at most) 100 time instants long the temperature of the system *Sys* oscillates in the real interval [2.9, 11.5] (after a short initial transitory phase):

$$\Box_{[0,100]}(global_clock \geq 5 \implies (temp \geq 2.9 \wedge temp \leq 11.5)).$$

More generally, our implementation of *Sys* satisfies the following three properties:

- $\Box_{[0,100]}(\neg deadlock)$, saying that the system does not deadlock;
- $\Box_{[0,100]}(safe)$, saying that the system does not violate the safety conditions;
- $\Box_{[0,100]}(\neg alarm)$: saying that the IDS does not fire any alarm.

The verification of these three properties requires around 15 min each, thanks to the underlying overapproximation.

In the next section, we will verify our CPS in the presence of three different cyber-physical attacks targeting either the sensor *sens* or the actuator *cool*. The reader can consult our models at <http://profs.scienze.univr.it/~merro/MODEST-FORTE/>.

5 A Static Security Analysis

In this section, we use the safety model checker `prohver` to test its limits when doing static security analysis of CPSs. In particular, we implement three simple cyber-physical attacks targeting our system *Sys*:

- a *DoS* attack on the actuation mechanism that may push the system to violate the safety conditions and hence in the invariant conditions;
- a *DoS* attack on the sensor that may deadlock the CPS without being noticed by the IDS;
- an *integrity* attack on the sensor, again undetected by the *IDS*, that may drive the CPS into a unsafe state but only for a limited period of time.

These attacks are implemented by tampering with either the physical devices (actuators and/or sensors) or the communication network (*man-in-the-middle*). In order to implement an attack on the sensor (*resp.*, actuator) we suppose the attacker is able to compromise the `Sensors()` (*resp.*, `Actuators()`) process.

Whereas the attacks targeting the communication network compromise either the `Proxy_sensor()` or the `Proxy_actuator()` process, depending whether they are targeting the sensor or the actuator. In general, attacks on the communication network do not require a deep knowledge on the physical dynamics of the CPS.

```

1 process E_Proxy_actuator(){
2   clock c;
3   do{ alt{ :: set_cool_on {= c = 0=};
4           invariant(c <= 0) when(c >= 0)
5           alt{ // drop the cool_on command in the time instant m
6               :: when(global_clock == m) tau
7               // in the other time instants forward correctly
8               :: when(global_clock < m || global_clock > m) cool_on_actuator
9           }
10          :: set_cool_off {= c = 0=};
11          invariant(c <= 0) when(c >= 0) cool_off_actuator
12        }
13      }
14    }

```

Fig. 8. DoS attack to the actuator

Attack 1. The first attack targets the actuator *cool* in a very simple manner. It operates exclusively in a specific time instant m , when it tries to drop the command to turn on the cooling system coming from the controller. Figure 8 shows the implementation of this man-in-the-middle attack compromising the `Proxy_actuator()` process.

We recall that the controller will turn on the cooling system only if it senses a temperature above 10 (as $\text{NOISE} = 0.1$, this means $\text{temp} > 9.9$). It is not difficult to see that this may happen only if $m > 7$ (in the time instant 7 the maximum temperature that may be reached by the engine is $7 \cdot (\text{DT} + \text{UNCERT}) = 7 \cdot (1 + 0.4) = 9.8^\circ$). Since the process `Ctrl()` never re-send commands to the actuator, if the attacker is successful in dropping the command to turn on the cooling system in the time slot m then the temperature will continue to rise, and after 2 time instants, in the time instant $m + 2$, the system will violate the safety conditions. This is noticed by the `IDS()` that will fire alarms every 5 time instants, until the CPS deadlocks because $\text{temp} > 20$.

We have verified the same properties stated in the previous section for the system *Sys* in isolation. None of those properties holds when the attack above operates in an instant $m > 7$. In particular, for $m > 7$ the system becomes unsafe in the time instant $m + 2$, and the `IDS()` detects the violation of the safety conditions with a delay of only 2 time instants. Summarising:

Attack 1: tested properties	$m \leq 7$	$m > 7$
$\square_{[0,100]}(\neg\text{deadlock})$	✓	✗
$\square_{[0,100]}(\text{safe})$	✓	✗
$\square_{[0,100]}(\neg\text{alarm})$	✓	✗
$\square_{[0,m+1]}(\text{safe})$	✓	✓
$\square_{[0,m+2]}(\text{safe})$	✓	✗
$\square_{[0,m+3]}(\neg\text{alarm})$	✓	✓
$\square_{[0,m+4]}(\neg\text{alarm})$	✓	✗

The properties above have been proved for all discrete time instants m , with $0 \leq m \leq 96$. The longest among these analyses required 20 min when overapproximating and at most 7 h when doing exact verification.

Attack 2. The second attack compromises the sensor in order to provide fake measurements to the controller. The compromised sensor operates as follows: (i) in any time instant smaller than or equal to 1 the sensor works correctly, (ii) in any time instant greater than 1 the sensor returns the temperature sensed at time 1. Figure 9 provides an implementation of the compromised sensor.

In the presence of this attack, the process `Ctrl()` will always detect a temperature below 10 and never activate the cooling system or the IDS. The system under attack will move to an unsafe state until the system invariant will be violated and the system will deadlock. Indeed, in the worst case scenario, after $\lceil \frac{9.9}{DT+UNCERT} \rceil = \lceil \frac{9.9}{1.4} \rceil = 8$ time instants the value of *temp* will be above 9.9° , and after further 4 time instants the system will violate the safety conditions. Furthermore, in the time instant $= \lceil \frac{20}{1.4} \rceil = 15$ the invariant may be broken and the system may deadlock because the state variable *temp* reaches 20.4° . This is a *lethal attack* as it causes a deadlock of the system. It is also a *stealthy attack* as it remains unnoticed until the end.

```

1 process E_Sensors(){
2   clock c;
3   do{
4     alt{ ::when(global_clock <= 1) //normal behaviour
5         req_sensor {= sens = any(z, z >= temp-NOISE && z <= temp+NOISE), c = 0 =};
6         invariant (c <= 0) when(c >= 0) ack_sensor
7     ::when(global_clock > 1) //attack
8         req_sensor {= c = 0 =}; //the measurement remains unchanged
9         invariant (c <= 0) when(c >= 0) ack_sensor
10    }
11 }

```

Fig. 9. DoS attack to the sensor

The results of our security analysis are summarised in the following table:

Attack 2: tested properties	
$\square_{[0,100]}(\neg alarm)$	✓
$\square_{[0,100]}(safe)$	✗
$\square_{[0,100]}(\neg deadlock)$	✗
$\square_{[0,11]}(safe)$	✓
$\square_{[0,12]}(safe)$	✗
$\square_{[0,14]}(\neg deadlock)$	✓
$\square_{[0,15]}(\neg deadlock)$	✗

The longest among these analyses required 35 min when overapproximating and at most 5 h when doing exact verification. Please, notice that this attack does not require any specific knowledge of the sensor device (such as the measurement equation). Thus, the same goal could be obtained by means of a man-in-the-middle attack that compromises the `Proxy_sensor()` process.

Attack 3. Our last attack is a variant of the previous one as it provides the controller with a temperature decreased by an offset (in this case 2), for n consecutive time instants. Unlike the previous attack, in case of encrypted communication, this attack cannot be mounted in the network as it requires the knowledge of the measurement equation. Figure 10 shows the implementation of a compromised sensor device acting as required. Basically, when $global_clock \leq n$ the compromised sensor returns a measurement affected by the offset; on the other hand, when $global_clock > n$ the sensor works correctly and returns the authentic measurement.

The effects of this attack on the system depends on its duration n .

- For $n \leq 7$ the attack is harmless as the variable `temp` may not reach a (critical) temperature above 9.9; thus, all properties seen for the system in isolation remain valid when the system is under attack.
- For $n = 8$, the variable `temp` might reach a temperature above 9.9 and the attack would delay the activation of the cooling system of one time instant. As a consequence, the system might get into an unsafe state in the time

```

1 process E_Sensors() {
2   clock c;
3   do { req_sensor {= c = 0 =};
4     invariant (c <= 0) when (c >= 0)
5     alt { :: when(global_clock <= n) //send corrupted measurement
6           {= sens = any(z, z >= (temp - 2 - NOISE) && z <= (temp - 2 + NOISE)),
7             c = 0 =};
8           :: when(global_clock > n) //send authentic measurement
9             {= sens = any(z, z >= (temp - NOISE) && z <= (temp + NOISE)), c = 0 =}
10          };
11     invariant (c <= 0) when (c >= 0) ack_sensor
12   }
13 }
```

Fig. 10. Integrity attack to the sensor device

instants 12 and 13, but no alarm will be fired (*stealthy attack*). This is proved by verifying the following properties:

- $\square_{[0,100]}((global_clock < 12 \vee global_clock > 14) \implies safe) \checkmark$
 - $\square_{[0,100]}((global_clock \leq 12 \wedge global_clock \geq 12) \implies safe) \times$
 - $\square_{[0,100]}((global_clock \leq 13 \wedge global_clock \geq 13) \implies safe) \times$
 - $\square_{[0,100]}(\neg alarm) \checkmark$.
- For $n > 8$ the system may get into an unsafe state in a time instant between 12 and $n + 12$. The IDS will fire the alarm but it will definitely miss a number of violations of safety conditions as after the instant $n + 6$ it does not fire any alarm, although we prove there are unsafe states. This is a *temporary attack* as the system behaves correctly after the time instant $n + 12$. Summarising:
- $\square_{[0,100]}(\neg deadlock) \checkmark$
 - $\square_{[0,100]}((global_clock < 12 \vee global_clock > n + 12) \implies safe) \checkmark$
 - $\square_{[0,100]}((global_clock \geq 12 \wedge global_clock \leq n + 12) \implies safe) \times$
 - $\square_{[0,100]}((global_clock > n + 6 \wedge global_clock \leq n + 12) \implies safe) \times$
 - $\square_{[0,100]}((global_clock < n + 1 \vee global_clock > n + 6) \implies \neg alarm) \checkmark$
 - $\square_{[0,100]}((global_clock \geq n + 1 \wedge global_clock \leq n + 6) \implies \neg alarm) \times$.

The properties above have been proved for all discrete time instants n , with $0 \leq n \leq 85$. The longest among these analyses required 1 h when overapproximating and at most 7 h when doing exact verification.

6 Conclusions

As said in the Introduction, the safety model checker within the MODEST TOOLSET relies on a modified version of the hybrid solver PHAVer, whose specification language is a slight variation of hybrid automata supporting compositional reasonings, where input and output variables are clearly distinguished [20]. Although, PHAVer would be a good candidate for the verification of small CPSs, we preferred to specify our case study in the high-level language HMODEST, supporting: (i) differential inclusion to model linear CPSs with constant bounded derivatives; (ii) linear formulae to express nondeterministic assignments within a dense interval; (iii) compositional programming style inherited from process algebra (*e.g.*, parallel composition, nondeterministic choice, loops, *etc.*); (iv) shared actions to synchronise parallel components.

In HMODEST, we have implemented a simple but totally realistic and nuanced cyber-physical system together with three cyber-physical attacks targeting the sensor or the actuator of the system. In particular, we have proposed: (i) a *DoS attack on the actuator* that operates as a man-in-the-middle on the connecting network; (ii) a *DoS attack on the sensor* that is achieved by compromising the sensor device; (iii) an *integrity attack on the sensor*, again by compromising the sensor device. Our implementation is quite clean and concise, although the current version of the language has still some problems in representing both instantaneous and delayed behaviours in an effective manner (we did not use the elegant

delay() construct as each instance introduces a new clock, with heavy implications on the verification performance). Furthermore, in order to verify our safety and invariant conditions we have implemented a `Safety()` process that is not really part of our CPS. From a designer point of view it would have been much more practical to use some kind of logic formula, such as: $\exists \diamond (\square_{[t, t+5]} temp > 9.9)$.

For the security analysis we have used the safety model checker `prohver`. Basically, we have verified LTL properties on the system under attack. Although, we have verified most of the properties that have been manually proved in [19], we have not been able to capture time properties on the responsiveness of the IDS to violations of the safety conditions. Properties such as:

- there are integers m and k such that the system may have an unsafe state at some instant $n > m$, and the IDS detects this violation with a delay of at least k time instants (k being a lower bound of the reaction time of the IDS);
- there is an instant n where the IDS fires an alarm but neither an unsafe state nor a deadlock occurs between the instants $n - k$ and $n + k$: this would provide a tolerance of the occurrence of *false positive*.

Note that `prohver` has been designed to do *probabilistic model-checking*, while in this paper we only do model checking. Actually, one of the reasons why we implemented our case study in HMODEST is because we aim at strengthening our security analysis by resorting to probabilistic model checking. This would allow us to replace nondeterministic uncertainty and nondeterministic noise with probability distributions (for instance, *normal distributions* are very common in this context).

A Comparison With Other Model-Checkers. We tried to verify our case study also with other model-checkers for distributed systems providing high-level specification languages and expressive query languages, such as PRISM [16], UPPAAL [2] and Real-Time Maude [22]. In particular, as our example has a discrete notion of time we started looking at verification tools supporting discrete time.

PRISM, for instance, relies on Markov decision processes or discrete-time Markov chains, depending whether one is interested in modelling nondeterminism or not. It supports the verification of both CTL and LTL properties (when dealing with nonprobabilistic systems). This allowed us to express the formula $\exists \diamond (\square_{[t, t+5]} temp > 9.9)$ to verify violations of the safety conditions, avoiding the implementation of the `Safety()` process. However, using integer variables to represent state variables with a fixed precision requires the introduction of extra transitions (to deal with nondeterministic errors) that significantly complicates the PRISM model.

In this respect, UPPAAL appears to be more efficient than PRISM, as we have been able to concisely express the error occurring in integer state variables thanks to the *select()* construct, in which the user can fix the granularity adopted to approximate a dense interval. This discrete representation provides an *under-approximation* of the system behaviour; thus, a finer granularity translates into an exponential increase of the complexity of the system, with obvious

consequences on the verification performance. UPPAAL has provided us with a simple way to implement the preemptive power of cyber-physical attacks by assigning priorities to processes. Thus, a system under attack can be easily represented by simply putting in parallel the system and the attacker. The tool supports the verification of a simplified version of CTL properties (no nesting of path formulae is allowed). Thus, as in HMODEST, we cannot express the formula $\exists \diamond (\Box_{[t, t+5]} temp > 9.9)$ and we had to implement a `Safety()` process.

Finally, we tried to model our case study in Real-Time Maude, a completely different framework for real-time systems, based on *rewriting logic*. The language supports object-like inheritance features that are quite helpful to represent complex systems in a modular manner. Communication channels have been used to implement our attacks on the physical devices. Furthermore, we used rational variables for a more concise discrete representation of state variables. We have been able to verify LTL and T-CTL properties, although the verification process resulted to be very slow due to a proliferation of rewriting rules when fixing a reasonable granularity to approximate dense intervals. As the verification logic is quite powerful, there is no need to implement the `Safety()` process.

Formal Methods for CPS Security. A few works use formal methods for CPS security, although they apply methods, and most of the time have goals, that are quite different from ours. As already said, the case study has been taken from [19]. In that paper the authors present a threat model for a formal study of a variety of cyber-physical attacks. They also propose a formal technique to assess the tolerance of CPSs to classes of attacks. The paper provides a stepping stone for formal and automated analysis techniques for checking the security of CPSs.

In [28, 29], Vigo presents an attack scenario that addresses some of the peculiarities of a cyber-physical adversary, and discussed how this scenario relates to other attack models popular in the security protocol literature. Unlike us, this paper focuses on DoS attacks without taking into consideration timing aspects. Rocchetto and Tippenhaur [25] introduce a taxonomy of the diverse attacker models proposed for CPS security and outline requirements for generalised attacker models; in [24], they then propose an extended Dolev-Yao attacker model suitable for CPS security. In their approach, physical layer interactions are modelled as abstract interactions between logical components to support reasoning on the physical-layer security of CPSs. This is done by introducing additional orthogonal channels. Time is not represented. Nigam et al. [21] work around the notion of Timed Dolev-Yao Intruder Models for Cyber-Physical Security Protocols by bounding the number of intruders required for the automated verification of such protocols. Following a tradition in security protocol analysis, they provide an answer to the question: How many intruders are enough for verification and where should they be placed? They also extend the strand space model to CPS protocols by allowing for the symbolic representation of time, so that they can use Real-Time Maude [22] along with SMT support. Their notion of time is however different from ours, as they focus on the time a message needs

to travel from an agent to another. The paper does not mention physical devices, such as sensors and/or actuators.

Acknowledgements. We thank the anonymous reviewers for their insightful and careful reviews that allowed us to significantly improve the paper. We thank Fabio Mogavero for stimulating discussions on model checking tools, and Arnd Hartmanns for “tips and tricks” on the MODEST TOOLSET. This work has been partially supported by the project “Dipartimenti di Eccellenza 2018–2022” funded by the Italian Ministry of Education, Universities and Research (MIUR).

References



1. Alur, R., Dill, D.L.: A theory of timed automata. *Theoret. Comput. Sci.* **126**(2), 183–235 (1994)
2. Behrmann, G., David, A., Larsen, K.G., Håkansson, J., Pettersson, P., Yi, W., Hendriks, M.: UPPAAL 4.0. In: D’Argenio, P., Miner, A., Rubino, G. (eds.) QEST 2006, pp. 125–126. IEEE Computer Society (2006). DOI: [10.1109/QEST.2006.59](https://doi.org/10.1109/QEST.2006.59)
3. Bogomolov, S., Frehse, G., Greitschus, M., Grosu, R., Pasareanu, C., Podelski, A., Strump, T.: Assume-guarantee abstraction refinement meets hybrid systems. In: Yahav, E. (ed.) HVC 2014. LNCS, vol. 8855, pp. 116–131. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13338-6_10
4. Bohnenkamp, H., Hermanns, H., Katoen, J.-P.: MOTOR: the MODEST tool environment. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 500–504. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_38
5. Chen, X., Abraham, E., Sankaranarayanan, S.: Flow*: an analyzer for non-linear hybrid systems. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 258–263. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_18
6. Duggirala, P.S., Mitra, S., Viswanathan, M., Potok, M.: C2E2: a verification tool for stateflow models. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 68–82. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_5
7. Falliere, N., Murchu, L., Chien, E.: W32.Stuxnet Dossier (2011)
8. Frehse, G.: Phaver Language Overview v0.35 (2006). http://www.verimag.imag.fr/~frehse/phaver_web/phaver_lang.pdf
9. Frehse, G.: Phaver: algorithmic verification of hybrid systems past hytech. *Int. J. Softw. Tools Technol. Transf.* **10**(3), 263–279 (2008)
10. Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_30
11. Hahn, E.M., Hartmanns, A., Hermanns, H., Katoen, J.: A compositional modelling and analysis framework for stochastic hybrid systems. *Formal Methods Syst. Des.* **43**(2), 191–232 (2013)

12. Hartmanns, A., Hermanns, H.: The modest toolset: an integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 593–598. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_51
13. Hennessy, M., Regan, T.: A process algebra for timed systems. *Inf. Comput.* **117**(2), 221–239 (1995)
14. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What’s decidable about hybrid automata? *J. Comput. Syst. Sci.* **57**(1), 94–124 (1998)
15. ICS-CERT: Cyber-Attack Against Ukrainian Critical Infrastructure. <https://ics-cert.us-cert.gov/alerts/IR-ALERT-H-16-056-01>
16. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
17. Lafferriere, G., Pappas, G.J., Sastry, S.: O-minimal hybrid systems. *Math. Contr. Sig. Syst.* **13**(1), 1–21 (2000)
18. Lanotte, R., Merro, M.: A semantic theory of the Internet of Things. *Inf. Comput.* **259**(1), 72–101 (2018)
19. Lanotte, R., Merro, M., Muradore, R., Viganò, L.: A formal approach to cyber-physical attacks. In: CSF 2017, pp. 436–450. IEEE Computer Society (2017). <https://doi.org/10.1109/CSF.2017.12>
20. Lynch, N.A., Segala, R., Vaandrager, F.W.: Hybrid I/O automata. *Inf. Comput.* **185**(1), 105–157 (2003)
21. Nigam, V., Talcott, C., Aires Urquiza, A.: Towards the automated verification of cyber-physical security protocols: bounding the number of timed intruders. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9879, pp. 450–470. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45741-3_23
22. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. *High. Order Symb. Comput.* **20**(1–2), 161–196 (2007)
23. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation-based abstraction refinement. *ACM Trans. Embed. Comput. Syst.* **6**(1), 8 (2007)
24. Rocchetto, M., Tippenhauer, N.O.: CPDY: extending the Dolev-Yao attacker with physical-layer interactions. In: Ogata, K., Lawford, M., Liu, S. (eds.) ICFEM 2016. LNCS, vol. 10009, pp. 175–192. Springer, Cham (2016). <https://doi.org/10.1007/978-3-319-47846-3>
25. Rocchetto, M., Tippenhauer, N.O.: On attacker models and profiles for cyber-physical systems. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9879, pp. 427–449. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45741-3_22
26. Roohi, N.: Remedies for building reliable cyber-physical systems. Ph.D. thesis, University of Illinois at Urbana-Champaign (2017)
27. Slay, J., Miller, M.: Lessons learned from the Maroochy Water Breach. In: Goetz, E., Shenoi, S. (eds.) ICCIP 2007. IIFIP, vol. 253, pp. 73–82. Springer, Boston, MA (2008). https://doi.org/10.1007/978-0-387-75462-8_6
28. Vigo, R.: The cyber-physical attacker. In: Ortmeier, F., Daniel, P. (eds.) SAFE-COMP 2012. LNCS, vol. 7613, pp. 347–356. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33675-1_31
29. Vigo, R.: Availability by design: a complementary approach to denial-of-service. Ph.D. thesis, Danish Technical University (2015)

30. Vladimerou, V., Prabhakar, P., Viswanathan, M., Dullerud, G.: STORMED hybrid systems. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfssdóttir, A., Walukiewicz, I. (eds.) ICALP 2008. LNCS, vol. 5126, pp. 136–147. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70583-3_12
31. Zacchia Lun, Y., D’Innocenzo, A., Malavolta, I., Di Benedetto, M.D.: Cyber-Physical Systems Security: a Systematic Mapping Study. CoRR abs/1605.09641 (2016). <http://arxiv.org/abs/1605.09641>



Relating Process Languages for Security and Communication Correctness (Extended Abstract)

Daniele Nantes¹✉  and Jorge A. Pérez²✉ 

¹ Universidade de Brasília, Brasília, Brazil
dnantes@unb.br

² University of Groningen and CWI, Amsterdam, The Netherlands
j.a.perez@rug.nl

Abstract. Process calculi are expressive specification languages for concurrency. They have been very successful in two research strands: (a) the analysis of *security protocols* and (b) the enforcement of correct *message-passing programs*. Despite their shared foundations, languages and reasoning techniques for (a) and (b) have been separately developed. Here we connect two representative calculi from (a) and (b): we encode a (high-level) π -calculus for multiparty sessions into a (low-level) applied π -calculus for security protocols. We establish the correctness of our encoding, and we show how it enables the integrated analysis of security properties *and* communication correctness by re-using existing tools.

1 Introduction

This paper connects two distinct formal models of communicating systems: a process language for the analysis of *security protocols* [12], and a process language for *session-based concurrency* [9, 10]. They are representative of two separate research strands:

- (a) Process models for security protocols, such as [12] (see also [7]), rely on variants of the applied π -calculus [1] to establish properties related to process execution (e.g., secrecy and confidentiality). These models support cryptography and term passing, but lack support for high-level communication structures.
- (b) Process models for session-based communication, such as [10] (see also [11]), use π -calculus variants equipped with type systems to enforce correct message-passing programs. Security extensions of these models target properties such as information flow and access control (cf. [2]), but usually abstract away from cryptography.

We present a *correct encoding* that connects two calculi from these two strands:

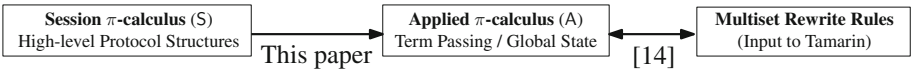
Work partially funded by FAP-DF 0193.001381/2017.

- **A**, a (low-level) applied π -calculus in which processes explicitly describe term communication, cryptographic operations, and state manipulation [12];
- **S**, a (high-level) π -calculus in which communication actions are organized as multiparty session protocols [5, 10].

Our aim is to exploit the complementary strenghts of **A** and **S** to analyze communicating systems that feature high-level communication structures (as in session-based concurrency [9, 10]) *and* use cryptographic operations and global state in protocol exchanges.

Our encoding of **S** into **A** describes how the structures typical of session-based, asynchronous concurrency can be compiled down, in a behavior-preserving manner, as process implementations in which communication of terms takes place exploiting rich equational theories and global state. To our knowledge, ours is the first work to relate process calculi for the analysis of communication-centric programs (**S**) and of security protocols (**A**), as developed in disjoint research strands.

We believe our results shed light on both (a) and (b). In one direction, they define a new way to reason about multiparty session processes. Process specifications in **S** can now integrate cryptographic operations and be analyzed by (re)using existing methods. In fact, since **A** processes can be faithfully translated into multiset rewriting rules using SAPIC [12] (which can in turn be fed into the Tamarin prover [14]), our encoding bridges the gap between **S** processes and toolsets for the analysis of security properties:



Interestingly, this connection can help to enforce communication correctness: we show how SAPIC/Tamarin can check *local formulas* representing local session types [10].

In the other direction, our approach allows us to enrich security protocol specifications with communication structures based on sessions. This is relevant because the analysis of security protocols is typically carried out on models such as, e.g., Horn clauses and rewriting rules, which admit efficient analysis but that lead to too low-level specifications. Our developments fit well in this context, as the structures intrinsic to session-based concurrency can conveniently describe communicating systems in which security protocols appear intertwined with higher-level interaction protocols.

This rest of the paper is organized as follows. Section 2 introduces the *Two-Buyer Contract Signing Protocol*, a protocol that is representative of the kind of systems that is hard to specify using **S** or **A** alone. Section 3 recalls the definitions of **S** and **A**, and also introduces S^* , which is a variant of **S** that is useful in our developments. Section 4 defines the encoding of **S** into **A**, using S^* as stepping stone, and establishes its correctness (Theorems 1, 2, and 3). Section 5 shows how our encoding can be used to reduce the enforcement of protocol conformance in **S** to the model checking of local formulas for **A** (Theorems 4 and 5). Section 6 revisits the Two-Buyer Contract Signing Protocol: we illustrate

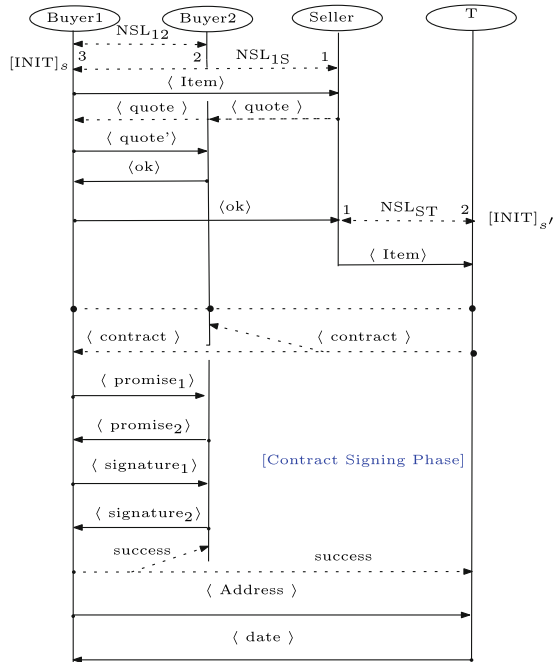


Fig. 1. The trusted Buyers-Seller protocol.

its process specification using S minimally extended with constructs from A , and show how key correctness properties can be mechanically verified using SAPIC/Tamarin. The paper closes by discussing related works and collecting concluding remarks (Sect. 7). Additional technical material and further examples are given in an appendix available online [15].

2 A Motivating Example: The Trusted Buyers-Seller Protocol

The *Trusted Buyers-Seller Protocol* extends the Two-Buyer Protocol [10], and proceeds in two phases. The first phase follows the global session type in [10], which offers a unified description of the way in which two buyers (B_1 and B_2) interact to purchase a book from a seller (S). In the second phase, once B_1 and B_2 agree in the terms of the purchase, the role of S is delegated to a *trusted third party* (T), which creates a contract for the transaction and collects the participants' signatures. This second phase relies on the *contract signing protocol* [8], which may resolve conflicts (due to unfulfilled promises from B_1 and B_2) and abort the conversation altogether. In this protocol, one key security property is *authentication*, which ensures that an attacker cannot impersonate B_i , S , or T . Relevant properties of communication correctness include *fidelity* and *safety*: while the former ensures that processes for B_i , S , and T follow the protocols specified by global/local types, the latter guarantees that such

processes do not get into errors at runtime. The protocol is illustrated in Fig. 1 and described next:

First Phase. B_1 , B_2 , and S start by establishing a session, after executing the Needham-Schroeder-Lowe (NSL) authentication protocol. Subsequently, they interact as follows:

1. B_1 sends the book title to S . Then, S replies back to both B_1 and B_2 the quote for the title. Subsequently, B_1 tells B_2 how much he can contribute.
2. If the amount is within B_2 's budget, then he accepts to perform the transaction, informs B_1 and S , and awaits the contract signing phase. Otherwise, if the amount offered by B_1 is not enough, B_2 informs S and B_1 his intention to abort the protocol.
3. Once B_1 and B_2 have agreed upon the purchase, S will *delegate* the session to the trusted party T , which will lead the contract signing phase. Upon completion of this phase, S (implemented by T) sends B_1 the delivery date for the book.

Second Phase. At this point, the trusted authority T , B_1 , and B_2 interact as follows:

4. T creates a new contract ct and a new memory cell s , useful to record information about the contract. T sends the contract ct to B_1 and B_2 for them to sign. T can start replying to the following requests: **success** (in case of successful communication), **abort** (request to abort the protocol), or **resolve** (request to solve a conflict).
5. Upon reception of contract ct from T , B_1 sends to B_2 his promise to sign it. Subsequently, B_1 expects to receive B_2 's promise:
 - If B_1 receives a valid response from B_2 , his promise is converted into a signature ($\langle\text{signature}_1\rangle$), which is sent back. Now, B_1 expects to receive a valid signature from B_2 : if this occurs, B_1 sends to T a **success** message; otherwise, B_1 sends T a **resolve** request, which includes the promise by B_2 and his own signature.
 - If B_1 does not receive a valid promise from B_2 , then B_1 asks T to cancel the purchase (an **abort** request), including his own promise ($\langle\text{promise}_1\rangle$) in the request.
6. Upon reception of contract ct from T , B_2 checks whether he obtained a valid promise from B_1 ; in that case, B_2 replies by sending his promise to sign it ($\langle\text{promise}_2\rangle$). Now, B_2 expects to receive B_1 's signature on ct : if the response is valid, B_2 sends its own signature ($\langle\text{signature}_2\rangle$) to B_1 ; otherwise, B_2 asks T to resolve. If B_2 does not receive a valid promise, then it aborts the protocol.

Clearly, S and A offer complementary advantages in modeling and analyzing the Trusted Buyers-Seller Protocol. On the one hand, S can represent high-level structures that are typical in the design of multiparty communication protocols. Such structures are essential in, e.g., the exchanges that follow session establishment in the first phase (which involves a step of session delegation to bridge with the second phase) and the handling of requests **success**, **abort** and **resolve** in

the second phase. Hence, S and its type-based verification techniques can be used to establish fidelity and safety properties. However, S is not equipped with constructs for directly representing cryptographic operations, as indispensable in, e.g., the NSL protocol for session establishment and in the exchanges of signatures/promises in the contract signing phase. The lack of these constructs prevents the formal analysis of authentication properties. On the other hand, A compensates for the shortcomings of S , for it can directly represent cryptographic operations on exchanged messages, as required to properly model the contract signing phase and, ultimately, to establish authentication. While A can represent the high-level communication structures mentioned above, it offers a too low-level representation of them, which makes reasoning about fidelity and safety more difficult than in S .

Our encoding from S into A , given in Sect. 4, will serve to combine the individual strengths of both languages. In Sect. 6, we will revisit this example: we will give a process specification using an extension of S with some constructs from A . This is consistent, because A is a low-level process language, and our encoding will define how to correctly compile S down to A (constructs from A will be treated homomorphically). Moreover, we will show how to use SAPIC/Tamarin to verify that implementations for B_1 , B_2 , S , and T respect their intended local types.

3 Two Process Models: A and S

3.1 The Applied π - Calculus (A)

Preliminaries. As usual in symbolic protocol analysis, messages are modelled by abstract terms (t, t', \dots) . We assume a countably infinite set of variables \mathcal{V} , a countably infinite set of names $\mathcal{N} = \text{PN} \cup \text{FN}$ (FN for fresh names, PN for public names), and a signature Σ (a set of function symbols, each with its arity).

We denote by \mathcal{T}_Σ the set of well-sorted terms built over Σ , \mathcal{N} , and \mathcal{V} . The set of ground terms (i.e., terms without variables) is denoted \mathcal{M}_Σ . A substitution is a partial function from variables to terms. We denote by $\sigma = \{t_1/x_1, \dots, t_n/x_n\}$ the substitution whose domain is $\text{Dom}(\sigma) = \{x_1, \dots, x_n\}$. We say σ is *grounding* for t if $t\sigma$ is ground. We equip the term algebra with an equational theory $=_E$, which is the smallest equivalence relation containing identities in E , a finite set of pairs the form $M = N$ where $M, N \in \mathcal{T}_\Sigma$, that is closed under application of function symbols, renaming of names, and substitution of variables by terms of the same sort. Furthermore, we require E to distinguish different fresh names, i.e., $\forall a, b \in \text{FN} : a \neq b \Rightarrow a \neq_E b$.

Given a set S , we write S^* and $S^\#$ to denote the sets of finite sequences of elements and of finite multisets of elements from S . We use the superscript $\#$ to annotate the usual multiset operations, e.g., $S_1 \cup^\# S_2$ denotes the union of multisets S_1, S_2 . Application of substitutions is extended to sets, multisets, and sequences as expected.

The set of *facts* is $\mathcal{F} := \{F(t_1, \dots, t_k) \mid t_i \in \mathcal{T}_\Sigma, F \in \Sigma_{fact} \text{ of arity } k\}$, where Σ_{fact} is an unsorted signature, disjoint from Σ . Facts will be used to

Table 1. Syntax of **A**: terms and processes.

$$\begin{aligned}
M, N &::= x, y \mid p \mid n \mid f(M_1, \dots, M_n) \quad (f \in \Sigma) \\
P, Q &::= \mathbf{0} \mid \mathbf{out}(M, N); P \mid \mathbf{in}(M, N); P \mid P \mid Q \mid !P \mid \nu n; P \mid \\
&\quad \mathbf{insert}((M, N)); P \mid \mathbf{delete} M; P \mid \mathbf{lookup} M \mathbf{as} x \mathbf{in} P \mathbf{else} Q \mid \\
&\quad \mathbf{lock} M; P \mid \mathbf{unlock} M; P \mid \mathbf{event} F; P \mid \mathbf{if} M = N \mathbf{then} P \mathbf{else} Q
\end{aligned}$$

annotate protocols (via events) and to define multiset rewrite rules. A fixed set of fact symbols will be used to encode the adversary’s knowledge, freshness information, and the messages on the network. The remaining fact symbols are used to represent the protocol state. For instance, fact $K(m)$ denotes that m is known by the adversary.

Syntax and Semantics. The grammar for terms (M, N) and processes (P, Q) , given in Table 1, follows [12]. In addition to usual operators for concurrency, replication, and name creation, the calculus **A** inherits from the applied π -calculus [1] input and output constructs in which terms appear both as communication subjects and objects. Also, **A** includes a conditional construct based on term equality, as well as constructs for reading from and updating an explicit *global state*:

- **insert** $((M, N)); P$ first binds the value N to a key M and then proceeds as P . Successive inserts may modify this binding; **delete** $M; P$ simply “undefines” the mapping for the key M and proceeds as P .
- **lookup** $M \mathbf{as} x \mathbf{in} P \mathbf{else} Q$ retrieves the value associated to M , binding it to variable x in P . If the mapping is undefined for M then the process behaves as Q .
- **lock** $M; P$ and **unlock** $M; P$ allow to gain and release exclusive access to a resource/key M , respectively, and to proceed as P afterwards. These operations are essential to specify parallel processes that may read/update a common memory.

Moreover, the construct **event** $F; P$ adds $F \in \mathcal{F}$ to a multiset of ground facts before proceeding as P . These facts will be used in the transition semantics for **A**, which is defined by a labelled relation between *process configurations* of the form $(\mathcal{E}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L})$, where: \mathcal{P} is a multiset of ground processes representing the processes executed in parallel; $\mathcal{E} \subseteq \text{FN}$ is the set of fresh names generated by the processes; $\mathcal{S} : \mathcal{M}_\Sigma \rightarrow \mathcal{M}_\Sigma$ is a partial function modeling stored information (state); σ is a ground substitution modeling the messages sent to the environment; and $\mathcal{L} \subseteq \mathcal{M}_\Sigma$ is the set of currently acquired locks. We write $\mathcal{S}(M) = \perp$ to denote that there is no information stored for M in \mathcal{S} . Also, notation $\mathcal{L} \setminus M$ stands for the set $\mathcal{L} \setminus \{M' \mid M' =_E M\}$.

We also require the notions of *frame* and a *deduction relation*. A frame $\nu \tilde{n}. \sigma$ consists of a set of fresh names \tilde{n} and a substitution σ : it represents the sequence of messages that have been observed by an adversary during a protocol execution and secrets \tilde{n} generated by the protocol, a priori unknown to the adversary.

Table 2. Deduction rules for A. In rule [App]: $\tilde{t} = (t_1, \dots, t_n)$.

$\frac{a \in (\text{FN} \cup \text{PN}) \setminus \tilde{n}}{\nu \tilde{n}. \sigma \vdash a}$	[Name] $\frac{\nu \tilde{n}. \sigma \vdash t \quad t =_E t'}{\nu \tilde{n}. \sigma \vdash t'}$	[Eq] $\frac{x \in \text{Dom}(\sigma)}{\nu \tilde{n}. \sigma \vdash x\sigma}$	[Frame] $\frac{\nu \tilde{n}. \sigma \vdash t_i}{\nu \tilde{n}. \sigma \vdash f\tilde{t}}$	[App]
---	--	--	--	-------

Table 3. Operational semantics for A.**Standard Operations**

$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{0\}, \sigma, \mathcal{L})$	\rightarrow_A	$(\mathcal{E}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L})$	
$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{P \mid Q\}, \sigma, \mathcal{L})$	\rightarrow_A	$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{P, Q\}, \sigma, \mathcal{L})$	
$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{!P\}, \sigma, \mathcal{L})$	\rightarrow_A	$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{!P, P\}, \sigma, \mathcal{L})$	
$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{\nu a; P\}, \sigma, \mathcal{L})$	\rightarrow_A	$(\mathcal{E} \cup \{a'\}, \mathcal{S}, \mathcal{P} \cup \# \{P\{a'/a\}\}, \sigma, \mathcal{L})$	C0
$(\mathcal{E}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L})$	$\xrightarrow{K(M)}_A$	$(\mathcal{E}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L})$	C1
$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{\text{out}(M, N); P\}, \sigma, \mathcal{L})$	$\xrightarrow{K(M)}_A$	$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{P\}, \sigma \cup \{N/x\}, \mathcal{L})$	C2
$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{\text{in}(M, N); P\}, \sigma, \mathcal{L})$	$\xrightarrow{K((M, N\tau))}_A$	$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{P\tau\}, \sigma, \mathcal{L})$	C3
$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{\text{out}(M, N); P, \text{in}(M', N'); Q\}, \sigma, \mathcal{L})$	\rightarrow_A	$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{P, Q\tau\}, \sigma, \mathcal{L})$	C4
$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{\text{if } M = N \text{ then } P \text{ else } Q\}, \sigma, \mathcal{L})$	\rightarrow_A	$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{P\}, \sigma, \mathcal{L})$	C5
$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{\text{if } M = N \text{ then } P \text{ else } Q\}, \sigma, \mathcal{L})$	\rightarrow_A	$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{Q\}, \sigma, \mathcal{L})$	C6
$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{\text{event } F; P\}, \sigma, \mathcal{L})$	\xrightarrow{F}_A	$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{P\}, \sigma, \mathcal{L})$	

Operations on Global State

$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{\text{insert}((M, N)); P\}, \sigma, \mathcal{L})$	\rightarrow_A	$(\mathcal{E}, \mathcal{S}[M \mapsto N], \mathcal{P} \cup \# \{P\}, \sigma, \mathcal{L})$	
$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{\text{delete } M; P\}, \sigma, \mathcal{L})$	\rightarrow_A	$(\mathcal{E}, \mathcal{S}[M \mapsto \perp], \mathcal{P} \cup \# \{P\}, \sigma, \mathcal{L})$	
$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{\text{lookup } M \text{ as } x \text{ in } P \text{ else } Q\}, \sigma, \mathcal{L})$	\rightarrow_A	$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{P\{V/x\}\}, \sigma, \mathcal{L})$	C7
$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{\text{lookup } M \text{ as } x \text{ in } P \text{ else } Q\}, \sigma, \mathcal{L})$	\rightarrow_A	$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{Q\}, \sigma, \mathcal{L})$	C8
$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{\text{lock } M; P\}, \sigma, \mathcal{L})$	\rightarrow_A	$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{P\}, \sigma, \mathcal{L} \cup \{M\})$	C9
$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{\text{unlock } M; P\}, \sigma, \mathcal{L})$	\rightarrow_A	$(\mathcal{E}, \mathcal{S}, \mathcal{P} \cup \# \{P\}, \sigma, \mathcal{L} \setminus M)$	

where:

C0: if a' fresh	C5: if $M =_E N$
C1: if $\nu \mathcal{E}. \sigma \vdash M$	C6: if $M \neq_E N$
C2: if x is fresh, $\nu \mathcal{E}. \sigma \vdash M$	C7: if $\exists N. N =_E M$ and $\mathcal{S}(N) =_E V$
C3: if $\exists \tau. \nu \mathcal{E}. \sigma \vdash M$ and $\nu \mathcal{E}. \sigma \vdash N\tau$ and τ grounding for N	C8: if $\forall N. N =_E M \Rightarrow \mathcal{S}(N) = \perp$
C4: if $M =_E M'$ and $\exists \tau. N =_E N'\tau$ and τ grounding for N'	C9: if $M \notin_E \mathcal{L}$

The deduction relation $\nu \tilde{n}. \sigma \vdash t$ models the adversary's ability to compute new messages from observed ones: it is the smallest relation between frames and terms defined by the rules in Table 2.

Transitions are of the form $(\mathcal{E}, \mathcal{S}, \mathcal{P}, \sigma, \mathcal{L}) \xrightarrow{\mathcal{F}}_A (\mathcal{E}', \mathcal{S}', \mathcal{P}', \sigma', \mathcal{L}')$, where \mathcal{F} is a set of ground facts (see Table 3). We write \rightarrow_A for $\xrightarrow{\emptyset}_A$ and \xrightarrow{f}_A for $\xrightarrow{\{f\}}_A$. As usual, \rightarrow_A^* denotes the reflexive, transitive closure of \rightarrow_A . Transitions denote either standard process operations or operations on the global state; they are sometimes denoted \rightarrow_{A_P} and \rightarrow_{A_S} , respectively.

Table 4. Process syntax and naming conventions for S.

$u ::= x \mid a$ (Identifiers)	$n ::= s \mid a$ (Names)	$e ::= v \mid x \mid e = e' \mid \dots$ (Expressions)
$c ::= s[\mathbf{p}] \mid x$ (Channels)	$v ::= a \mid \text{true} \mid \text{false} \mid s[\mathbf{p}]$ (Values)	
$m ::= (\mathbf{q} \triangleright \mathbf{p} : v) \mid (\mathbf{q} \triangleright \mathbf{p} : c) \mid (\mathbf{q} \triangleright \mathbf{p} : l)$ (Messages)		
$P ::= \bar{u}[\mathbf{p}](y).P$ (Req)	$c!\langle \mathbf{p}, c \rangle.P$ (Deleg)	$P \mid Q$ (Parallel)
$\mid u[\mathbf{p}](y).P$ (Acc)	$c?(\langle \mathbf{q}, y \rangle).P$ (Recep)	$\mathbf{0}$ (Inaction)
$\mid c!\langle \mathbf{p}, e \rangle.P$ (Send)	$c \oplus \langle \mathbf{p}, l \rangle.P$ (Select)	$(\nu n)P$ (N.Hiding)
$\mid c?(\mathbf{p}, x).P$ (Recv)	$c \&(\mathbf{p}, \{l_i : P_i\}_{i \in I})$ (Branch)	$s[\tilde{\mathbf{p}}] : h$ (M. Queue)
	$\mid \text{if } e \text{ then } P \text{ else } Q$ (Condit.)	$h ::= h \cdot m \mid \emptyset$ (Queue)

3.2 Multiparty Session Processes (S)

Syntax. The syntax of *processes*, ranged over by P, Q, \dots and that of *expressions*, ranged over by e, e', \dots , is given by the grammar of Table 4, which also shows name conventions. We assume two disjoint countable set of names: one ranges over *shared names* a, b, \dots and another ranges over *session names* s, s', \dots . Variables range over x, y, \dots ; *participants* (or *roles*) range over the naturals and are denoted as $\mathbf{p}, \mathbf{q}, \mathbf{p}', \dots$; *labels* range over l, l', \dots and *constants* range over $\text{true}, \text{false}, \dots$. We write $\tilde{\mathbf{p}}$ to denote a finite sequence of participants $\mathbf{p}_1, \dots, \mathbf{p}_n$ (and similarly for other elements). Given a session name s and a participant \mathbf{p} , we write $s[\mathbf{p}]$ to denote a (*session*) *endpoint*.

The intuitive meaning of processes is as in [5, 10]. The processes $\bar{u}[\mathbf{p}](y).P$ and $u[\mathbf{p}](y).Q$ can respectively request and accept to initiate a session through a shared name u . In both processes, the bound variable y is the placeholder for the channel that will be used in communications. After initiating a session, each channel placeholder will be replaced by an endpoint of the form $s[\mathbf{p}_i]$ (i.e., the runtime channel of \mathbf{p}_i in session s). Within an established session, process may send and receive basic values or session names (*session delegation*) and select and offer labeled, deterministic choices (cf. constructs $c \oplus \langle \mathbf{p}, l \rangle.P$ and $c \&(\mathbf{p}, \{l_i : P_i\}_{i \in I})$). The input/output operations (including delegation) specify the channel and the sender or the receiver, respectively.

Message queues model asynchronous communication. A message $(\mathbf{p} \triangleright \mathbf{q} : v)$ indicates that \mathbf{p} has sent a value v to \mathbf{q} . The empty queue is denoted by \emptyset . By $h \cdot m$ we denote the queue obtained by concatenating message m to the queue h . By $s[\tilde{\mathbf{p}}] : h$ we denote the queue h of the session s initiated between participants $\tilde{\mathbf{p}} = \mathbf{p}_1, \dots, \mathbf{p}_n$; when the participants are clear from the context we shall write $s : h$ instead of $s[\tilde{\mathbf{p}}] : h$.

Request/accept actions bind channel variables, value receptions bind value variables, channel receptions bind channel variables, hidings bind shared and session names. In $(\nu s)P$ all occurrences of $s[\mathbf{p}]$ and queue s inside P are bound. We denote by $fn(Q)$ the set of free names in Q . A process is *closed* if it does not contain free variables or free session names. Unless stated otherwise, we only consider closed processes.

Table 5. Structural congruence for \mathbf{S} processes.

$P \mid \mathbf{0} \equiv P$	$P \mid Q \equiv Q \mid P$	$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	$(\nu a)\mathbf{0} \equiv \mathbf{0}$	$(\nu s)(s : \emptyset) \equiv \mathbf{0}$
$(\nu r)P \mid Q \equiv (\nu r)(P \mid Q)$, if $r \notin \text{fn}(Q)$		$(\nu r)(\nu r')P \equiv (\nu r')(\nu r)P$, where $r ::= a \mid s$		
$s[\tilde{p}] : h \cdot (q \triangleright p : \zeta) \cdot (q' \triangleright p' : \zeta') \cdot h' \equiv s[\tilde{p}] : h \cdot (q' \triangleright p' : \zeta') \cdot (q \triangleright p : \zeta) \cdot h'$, if $p \neq p'$ or $q \neq q'$				

Table 6. Reduction rules for \mathbf{S} (Rule [If-F] omitted).

$a[\tilde{p}_1](y)P_1 \mid \dots \mid a[\tilde{p}_{n-1}](y)P_{n-1} \mid \bar{a}[\tilde{p}_n](y).P_n \longrightarrow_S$	[Init]
$(\nu s)(P_1\{s[\tilde{p}_1]/y\} \mid \dots \mid P_{n-1}\{s[\tilde{p}_{n-1}]/y\} \mid P_n\{s[\tilde{p}_n]/y\} \mid s[\tilde{p}] : \emptyset)$	
$s[\tilde{p}]!(\langle q, e \rangle).P \mid s : h \longrightarrow_S P \mid s : h \cdot (p \triangleright q : v)$ ($e \downarrow v$)	[Send]
$s[\tilde{p}]!(\langle q, s'[\tilde{p}'] \rangle).P \mid s : h \longrightarrow_S P \mid s : h \cdot (p \triangleright q : s'[\tilde{p}'])$	[Deleg]
$s[\tilde{p}] \oplus \langle q, l \rangle . P \mid s : h \longrightarrow_S P \mid s : h \cdot (p \triangleright q : l)$	[Sel]
$s[\tilde{p}]?(q, x).P \mid s : (q \triangleright p : v) \cdot h \longrightarrow_S P\{v/x\} \mid s[\tilde{p}] : h$	[Recv]
$s[\tilde{p}]?(q, y).P \mid s : (q \triangleright p : s'[\tilde{p}']) \cdot h \longrightarrow_S P\{s'[\tilde{p}']/y\} \mid s[\tilde{p}] : h$	[SRecv]
$s[\tilde{p}] \&(q, \{l_i : P_i\}_{i \in I} \mid s : (q \triangleright p : l_j) \cdot h \longrightarrow_S P_j \mid s : h$ ($j \in I$)	[Branch]
$\text{if } e \text{ then } P \text{ else } Q \longrightarrow_S P$ ($e \downarrow \text{true}$)	[If-T]
$P \equiv P' \text{ and } P' \longrightarrow_S Q' \text{ and } Q \equiv Q' \Rightarrow P \longrightarrow_S Q$	[Str]
$P \longrightarrow_S P' \Rightarrow E[P] \longrightarrow_S E[P']$	[Ctx]

Semantics. \mathbf{S} processes are governed by a reduction semantics, which relies on a *structural congruence* relation, denoted \equiv and defined by adding α -conversion to the rules of Table 5. Reduction rules are given in Table 6; we write $P \longrightarrow_S P'$ for a reduction step. We rely on the following syntax for contexts: $E ::= [] \mid P \mid (\nu a)E \mid (\nu s)E \mid E \mid E$.

We briefly discuss the reduction rules. Rule [Init] describes the initiation of a new session among n participants that synchronize over the shared name a . After session initiation, the participants will share a private session name (s in the rule), and an empty queue associated to it ($s[\tilde{p}] : \emptyset$ in the rule). Rules [Send], [Deleg] and [Sel] add values, channels and labels, respectively, into the message queue; in Rule [Send], $e \downarrow v$ denotes the evaluation of the expression e into a value v . Rules [Recv], [SRecv] and [Branch] perform complementary de-queuing operations. Other rules are self-explanatory.

3.3 The Calculus \mathbf{S}^*

We now introduce \mathbf{S}^* , a variant of \mathbf{S} which will simplify the definition of our encoding into \mathbf{A} . The syntax of \mathbf{S}^* processes is as follows:

$$\begin{aligned}
P, Q ::= & \mathbf{0} \mid \bar{u}[\tilde{p}](\tilde{y}).P \mid u[\tilde{p}](\tilde{y}).P \mid P \mid Q \mid (\nu n)P \mid \text{if } e \text{ then } P \text{ else } Q \\
& \mid c_{pq}!(e : \text{msg}).P \mid c_{pq}?(y).P \mid c_{pq}?(x).P \mid c_{pq}!(\langle c'_{p'q'} : \text{chan} \rangle).P \mid \\
& \mid c_{pq} \oplus \langle l : \text{lbl} \rangle . P \mid c_{pq} \&(\{l_i : P_i\}_{i \in I} \mid s_{pq} : h
\end{aligned}$$

where c_{pq} denotes a channel annotated with participant identities, $h ::= h \cdot m \mid \emptyset$ and $m ::= \langle \text{msg}, v \rangle \mid \langle \text{chan}, s_{pq} \rangle \mid \langle \text{lbl}, l \rangle$. The main differences between \mathbf{S} and \mathbf{S}^* are:

- Intra-session communication relies on annotated channels, and output prefixes include a *sort* for the communicated messages (**msg** for values, **chan** for delegated sessions, **lbl** for labels).
- While S uses a single queue per session, in S^* for each pair of participants there will be two queues, one in each direction. This simplifies the definition of structural congruence \equiv for S^* , which results from that for S as expected and is omitted.
- Constructs for session request and acceptance in S^* depend on a sequence of variables, rather than on a single variable. In these constructs, denoted $\bar{u}[p](\tilde{y}).P$ and $u[p](\tilde{y}).P$, respectively, \tilde{y} is a sequence of variables of the form y_{pq} , for some p, q .

With these differences in mind, the reduction semantics for S^* , denoted \longrightarrow_{S^*} , follows that for S (Table 6). Reduction rules for S^* include the following:

$$\begin{array}{l}
a[1](\tilde{y}_1).P_1 \mid \dots \mid a[n-1](\tilde{y}_{n-1}).P_{n-1} \mid \bar{a}[n](\tilde{y}_n).P_n \longrightarrow_{S^*} \quad [\text{Init}^*] \\
(\nu s)(P_1\{s/y\} \mid \dots \mid P_{n-1}\{s/y\} \mid P_n\{s/y\} \mid \tilde{y}_1\{s/y\} : \emptyset \mid \dots \mid \tilde{y}_n\{s/y\} : \emptyset) \\
y_{pq}!\langle e : \mathbf{msg} \rangle.P \mid y_{pq} : h \longrightarrow_{S^*} P \mid y_{pq} : h \cdot \langle \mathbf{msg}, v \rangle \quad (e \downarrow v) \quad [\text{Send}^*] \\
y_{pq}?(x).P \mid y_{qp} : \langle \mathbf{msg}, v \rangle \cdot h \longrightarrow_{S^*} P\{v/x\} \mid y_{qp} : h \quad [\text{Recv}^*]
\end{array}$$

Notice that in Rule [Init*], we only need to write $P_i\{s/y\}$: after reduction, these variables will be of the form s_{pq} . In that rule, each $\tilde{y}_i\{s/y\} : \emptyset$ denotes several queues (one for each name $y_{pq} \in \tilde{y}_i$), rather than a single queue.

It is straightforward to define an auxiliary encoding $([\cdot]) : S \mapsto S^*$. For instance:

$$\begin{array}{ll}
([s[p]!\langle q, e \rangle.P]) = s_{pq}!\langle e : \mathbf{msg} \rangle.([P]) & ([s[p]?(q, x).P]) = s_{qp}?(x).([P]) \\
([s[p]!\langle q, z_{p'} \rangle.P]) = s_{pq}!\langle \langle z_{p'} : \mathbf{chan} \rangle \rangle.([P]) & ([s[p]?(q, x).P]) = s_{qp}?(x).([P])
\end{array}$$

The full encoding, given in [15], enjoys the following property:

Theorem 1. *Let $P \in S$. Then: (a) If $P \longrightarrow_S P'$, then $([P]) \longrightarrow_{S^*} ([P'])$. (b) If $([P]) \longrightarrow_{S^*} R$, then there exists $P' \in S$ such that $P \longrightarrow_S P'$ and $([P']) = R$.*

Given the encoding $([\cdot]) : S \mapsto S^*$ and Theorem 1 above, we now move on to define an encoding $([\cdot]) : S^* \mapsto A$. By composing these encodings (and their correctness results—Theorems 2 and 3), we will obtain a behavioral-preserving compiler of S into A .

4 Encoding S^* into A

We now present our encoding $([\cdot]) : S^* \mapsto A$ and establish its correctness. The encoding is defined in Table 7; it uses the set of facts $F_S = \{\text{honest, sndnonce, rcvnonce, sndchann, rcvchann, out, inp, dels, recs, sel, bra, close}\}$. Facts will be used as event annotations in process executions, and also for model checking communication correctness via trace formulas in the following section. Our encoding will rely on the equational theory for **pairing**, which is embedded in Tamarin prover [14], and includes function symbols $\langle _, _ \rangle$, **fst** and **snd**, for

Table 7. Encoding from S^* to A .

Implementing Session Establishment
$\llbracket \bar{a}[3](\tilde{y}_3).P \rrbracket = \nu s; P_{31}; P_{32}; \mathbf{insert}(\langle \tilde{s}_{ij}, \emptyset \rangle); \mathbf{event\ init}(\tilde{s}_{ij});$ $\mathbf{event\ sndchann}(pk(ska_{31}), pk(y_1), s); \mathbf{out}(u_1, s);$ $\mathbf{event\ sndchann}(pk(ska_{32}), pk(y_2), s); \mathbf{out}(u_2, s); \llbracket P \rrbracket$ $P_{3i} = \nu ska_{3i}; \mathbf{out}(c, pk(ska_{3i})); \mathbf{event\ honest}(pk(ska_{3i})); \mathbf{in}(c, pk(y_i));$ $\nu n_{3i}; \mathbf{event\ sndnonce}(pk(ska_{3i}), pk(y_i), aenc(\langle n_{3i}, pk(ska_{3i}) \rangle, pk(y_i)))$ $\mathbf{out}(c, aenc(\langle n_{3i}, pk(ska_{3i}) \rangle, pk(y_i))); \mathbf{in}(c, aenc(\langle n_{3i}, u_i, pk(y_i) \rangle, pk(ska_{3i})));$ $\mathbf{event\ rcvnonce}(pk(y_i), pk(ska_{3i}), aenc(\langle n_{3i}, u_i, pk(y_i) \rangle, pk(ska_{3i})))$
$\llbracket a[i](\tilde{y}_i).P \rrbracket = \nu ska_i; \mathbf{in}(c, pk(x_i)); \mathbf{event\ honest}(pk(ska_i)); \mathbf{in}(c, aenc(\langle y, pk(x_i) \rangle, pk(ska_i)));$ $\mathbf{event\ rcvnonce}(pk(x_i), pk(ska_i), aenc(\langle y, pk(x_i) \rangle, pk(ska_i)))$ $\nu n_i; \mathbf{event\ sndnonce}(pk(ska_i), pk(x_i), aenc(\langle y, n_i, pk(ska_i) \rangle, pk(x_i)))$ $\mathbf{out}(c, aenc(\langle y, n_i, pk(ska_i) \rangle, pk(x_i))); \mathbf{in}(n_i, z);$ $\mathbf{event\ rcvchann}(pk(x_i), pk(ska_i), z); \llbracket P \rrbracket$
Implementing Intra-Session Communication
$\llbracket c_{pq}!(e : \mathbf{msg}).P \rrbracket = \mathbf{lock\ } c_{pq}; \mathbf{lookup\ } c_{pq} \mathbf{ as\ } x \mathbf{ in\ } (\mathbf{insert}(\langle c_{pq}, x \cdot \langle \mathbf{msg}, v \rangle \rangle);$ $\mathbf{event\ out}(c_{pq}, v); \mathbf{unlock\ } c_{pq}; \llbracket P \rrbracket \quad e \downarrow v$ $\llbracket c_{pq}?(x).P \rrbracket = \mathbf{lock\ } c_{pq}; \mathbf{lookup\ } c_{pq} \mathbf{ as\ } z_v \mathbf{ in\ } (\mathbf{if\ fst}(z_v) = \langle \mathbf{msg}, z \rangle \mathbf{ then}$ $(\mathbf{insert}(\langle c_{pq}, \mathbf{snd}(z_v) \rangle); \mathbf{event\ inp}(c_{pq}, \mathbf{fst}(z_v))); \mathbf{unlock\ } c_{pq}; \llbracket P\{z/x\} \rrbracket)$
$\llbracket c_{pq}!(\langle c' : \mathbf{chan} \rangle).P \rrbracket = \mathbf{lock\ } c_{pq}; \mathbf{lookup\ } c_{pq} \mathbf{ as\ } x \mathbf{ in\ } (\mathbf{insert}(\langle c_{pq}, x \cdot \langle \mathbf{chan}, c' \rangle \rangle);$ $\mathbf{event\ dels}(c_{pq}, c'); \mathbf{unlock\ } c_{pq}; \llbracket P \rrbracket$ $\llbracket c_{pq}?(x).P \rrbracket = \mathbf{lock\ } c_{pq}; \mathbf{lookup\ } c_{pq} \mathbf{ as\ } z_v \mathbf{ in\ } (\mathbf{if\ fst}(z_v) = \langle \mathbf{chan}, z \rangle \mathbf{ then}$ $(\mathbf{insert}(\langle c_{pq}, \mathbf{snd}(z_v) \rangle); \mathbf{event\ recs}(c_{pq}, \mathbf{fst}(z_v))); \mathbf{unlock\ } c_{pq}; \llbracket P\{z/x\} \rrbracket)$
$\llbracket c_{pq} \oplus \langle l : \mathbf{lbl} \rangle.P \rrbracket = \mathbf{lock\ } c_{pq}; \mathbf{lookup\ } c_{pq} \mathbf{ as\ } x \mathbf{ in\ } (\mathbf{insert}(\langle c_{pq}, x \cdot \langle \mathbf{lbl}, l \rangle \rangle);$ $\mathbf{event\ sel}(c_{pq}, l); \mathbf{unlock\ } c_{pq}; \llbracket P \rrbracket$
$\llbracket c_{pq} \& \langle \{l_i : P_i\} \rrbracket = \mathbf{lock\ } c_{pq}; \mathbf{lookup\ } c_{pq} \mathbf{ as\ } z_l \mathbf{ in\ } (\mathbf{if\ fst}(z_l) = \langle \mathbf{lbl}, l_1 \rangle \mathbf{ then}$ $\mathbf{insert}(\langle c_{pq}, \mathbf{snd}(z_l) \rangle); \mathbf{event\ bra}(c_{pq}, l_1); \mathbf{unlock\ } c_{pq}; \llbracket P_1 \rrbracket$ $\mathbf{else\ if\ fst}(z_l) = \langle \mathbf{lbl}, l_2 \rangle \mathbf{ then}$ $\mathbf{insert}(\langle c_{pq}, \mathbf{snd}(z_l) \rangle); \mathbf{event\ bra}(c_{pq}, l_2); \mathbf{unlock\ } c_{pq}; \llbracket P_2 \rrbracket)$
$\llbracket \mathbf{0} \rrbracket = \mathbf{event\ close} \quad \llbracket s[\tilde{p}] : h \rrbracket = \mathbf{0}$
$\llbracket (\nu s)P \rrbracket = \nu s; \llbracket P \rrbracket \quad \llbracket P \mid Q \rrbracket = \llbracket P \rrbracket \mid \llbracket Q \rrbracket \quad \llbracket \mathbf{if\ } e \mathbf{ then\ } P \mathbf{ else\ } Q \rrbracket = \mathbf{if\ } e \mathbf{ then\ } \llbracket P \rrbracket \mathbf{ else\ } \llbracket Q \rrbracket$

pairing and projection of first and second parameters of a pair. Communication within a secure established session is expressed by the manipulation of queues, which will be stored in the set of states \mathcal{S} . In SAPIC, we implement queues y_{pq} and y_{qp} as $q(y, \mathbf{p}, \mathbf{q})$ and $q(y, \mathbf{q}, \mathbf{p})$, respectively, where q is a function symbol for queues. Also, $s_{pq} : \emptyset$ is implemented as $\mathbf{insert}(\langle s_{pq}, \mathbf{init} \rangle)$.

Session Initiation. The (high-level) mechanism of session initiation of Rule [Init] in S^* (Table 6) is implemented in A by following the Needham-Schroeder-Lowe (NSL) authentication protocol [13]; see Table 7 (top). We use NSL because it is simple, and it has already been formalized in SAPIC. For simplicity,

we present the implementation for three participants; the extension to n participants is as expected. The encoding creates queues for intra-session communication using processes $\text{insert}(\langle \widetilde{s}_{ij}, \emptyset \rangle)$. The security verification uses the built-in library `asymmetric-encryption` available in Tamarin [14], and assumes the usual signature and equational theory for public keys pk , secret keys sk , asymmetric encryption $aenc$ and decryption dec .

Intra-session Communication. Process $\llbracket c_{pq}!(e : \text{msg}).P \rrbracket$ first acquires a lock in the queue c_{pq} to avoid interference. Then, a `lookup_as_` process checks the state of c_{pq} and enqueues message $\langle \text{msg}, v \rangle$ at its end. Finally, the encoding signals this operation by executing `event out`(c_{pq}, v) before unlocking c_{pq} and proceeding as as $\llbracket P \rrbracket$. The encoding of session delegation $\llbracket c_{pq}!\langle c : \text{chan} \rangle.P \rrbracket$ is very similar: the only differences are the sort of the communicated object and the event signaled at the end (`dels`(c_{pq}, c')).

As above, process $\llbracket c_{pq}?(x).P \rrbracket$ first acquires a lock and checks the queue c_{qp} . If it is of the form $\langle \text{msg}, - \rangle$ then it stores it in a variable z_v : it consumes the first part ($\text{fst}(z_v)$) and updates c_{qp} with the second part. The implementation then signals an event `inp`(c_{pq}, z_v) before unlocking c_{qp} and proceeding as $\llbracket P \rrbracket$. Process $\llbracket c_{pq}?(x).P \rrbracket$ (reception of a delegated session) is similar; in this case, the queue should contain a value of sort `chan` and the associated event is `recs`($c_{pq}, \text{fst}(z_v)$).

Process $\llbracket \mathbf{0} \rrbracket$ simply executes an event `close`. In the prototype SAPIC implementation of our encoding, this event mentions the name of the corresponding session c_{qp} .

Finally, process $\llbracket c_{pq} : h \rrbracket$ is $\mathbf{0}$ because we implement queues using the global state in **A**. The implementation of the remaining constructs in **A** is self-explanatory.

Remark 1. Since our encoding operates on *untyped* processes, we could have sort mismatches in queues (cf. Rule [If-F]). To avoid this, encodings of input-like processes (e.g., $s_{pq}?(x).P$), use the input of a *dummy* value that allows processes to reduce.

Correctness of $\llbracket \cdot \rrbracket$. We first associate to each ground process $P \in \mathbf{S}^*$ a process configuration via the encoding in Table 7. Below we assume that \tilde{s} , I , and I' may be empty, allowing the encoding of communicating processes (obtained after session initiation); we also assume that the set of (free) variables in P (denoted $\text{var}(P)$) can be instantiated with ground terms that can be deduced from the current frame.

Definition 1 *Suppose an \mathbf{S}^* process $R \equiv (\nu s)(\prod_{i \in I} P_i \mid \prod_{j,k \in I'} s_{pqk} : h_{j,k})$, with $\text{var}(R) = \{x_1, \dots, x_n\}$. A process configuration for R , denoted $C[\llbracket R \rrbracket]$, is defined as:*

$$(\mathcal{E} \cup \{s\}, \mathcal{S} \cup \{s_{pqk} : h_{j,k} \mid j, k \in I'\}, \left\{ \prod_{i \in I} \llbracket P_i \rrbracket \right\}, \sigma, \mathcal{L}),$$

where $\text{var}(R) \subseteq \text{dom}(\sigma)$ and σ is grounding for x_i , $i = 1, \dots, n$.

With some abuse of notation we say that C is a process configuration for R . Observe that different process configurations C, C', \dots can be associated to a same process $R \in \mathsf{S}$ once one considers variations of $\mathcal{E}, \mathcal{S}, \sigma, \mathcal{L}$.

Theorem 2 (Completeness). *Let $P \in \mathsf{S}^*$. If $P \longrightarrow_{\mathsf{S}^*} P'$ then for all process configuration C , there exists a process configuration C' such that $C[[P]] \longrightarrow_{\mathsf{A}}^* C'[[P']]$.*

Proof. The proof is by structural induction, analyzing the rule applied in $P \longrightarrow_{\mathsf{S}^*} P'$ via encoding in Table 7 and the rules in Table 3. See [15] for details. \square

To prove soundness, we rely on a Labeled Transition System for S^* , denoted $P \xrightarrow{\lambda} P'$. Such an LTS, and the proof of the theorem below, can be found in [15].

Theorem 3 (Soundness). *Let $P \in \mathsf{S}^*$ and C be such that $C[[P]] \longrightarrow_{\mathsf{A}_P} R$. Then there exist $P' \in \mathsf{S}^*$, a C' , and λ such that $R \longrightarrow_{\mathsf{A}}^* C'[[P']]$ and $P \xrightarrow{\lambda} P'$.*

5 Multiparty Session Types and Their Local Formulas

Using (\cdot) and $[[\cdot]]$, in this section we connect well-typedness of processes in S [10] with the satisfiability of *local formulas*, which model the execution of A processes.

5.1 Global and Local Types

Rather than defining multiparty session types for A processes, we would like to model checking local types by re-using existing tools for A : SAPIC [12] and Tamarin [14]. Concretely, next we shall connect typability for S processes with satisfiability for A processes. To formalize these results, we first recall some essential notions for multiparty session types; the reader is referred to [5, 10] for an in-depth presentation.

Global types G, G' describe multiparty session protocols from a vantage point; they offer a complete perspective on how two or more participants should interact. On the other hand, *local (session) types* T, T' describe how each participant contributes to the multiparty protocol. A *projection function* relates global and local types: the projection of G onto participant n is denoted $G|_n$. The syntax for global and local types, given in Table 8 is standard [10]. A complete description of session types can be found in [15].

Example 1. Figure 2 gives three global types for the protocol in Sect. 2: while G_{init} represents the first phase, both G_{contract} and G_{sign} are used to represent the second. In G_{sign} , we use G_{resolve_i} to denote a global protocol for resolving conflicts; see [15] for details.

Table 8. Global and local types [10].

$S ::= \text{bool} \mid \text{nonce} \mid \text{msg} \mid \text{temp} \mid \dots \mid G$	Sorts	$U ::= S \mid T$	Exchange Types												
(Global Types) $G ::= \mathbf{p} \rightarrow \mathbf{q} : \langle U \rangle . G \mid \mathbf{p} \rightarrow \mathbf{q} : \{l_i : G_i\}_{i \in I} \mid \mathbf{end}$															
(Local Types) $T ::= !\langle \mathbf{p}, U \rangle . T \mid ?\langle \mathbf{p}, U \rangle . T \mid \oplus \langle \mathbf{p}, \{l_i : T_i\} \rangle \mid \&\langle \mathbf{p}, \{l_i : T_i\} \rangle \mid \mathbf{end}$															
<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 15%; vertical-align: top; padding: 5px;">$G_{\text{init}} :$</td> <td style="width: 45%; padding: 5px;"> $(I.1) \ 3 \rightarrow 1 : \langle \text{Title} \rangle$ $(I.2) \ 1 \rightarrow \{2, 3\} : \langle \text{quote} \rangle$ $(I.3) \ 3 \rightarrow 2 : \langle \text{quote}' \rangle$ $(I.4) \ 2 \rightarrow \{1, 3\} : \begin{cases} \text{ok} : G_{\text{contract}} \\ \neg\text{ok} : \mathbf{end} \end{cases}$ </td> <td style="width: 15%; vertical-align: top; padding: 5px;">$G_b :$</td> <td style="width: 25%; padding: 5px;"> $(1') \ 1 \rightarrow 2 : \langle T \rangle$ $T = (G_{\text{contract}}) _1$ </td> </tr> <tr> <td style="vertical-align: top; padding: 5px;">$G_{\text{contract}} :$</td> <td style="padding: 5px;"> $(c.1) \ 1 \rightarrow \{2, 3\} : \langle \text{contract} \rangle$ $(c.2) \ 3 \rightarrow 2 : \langle \text{promise} \rangle$ $(c.3) \ 2 \rightarrow 3 : \begin{cases} \text{ok} : 2 \rightarrow 3 : \langle \text{promise} \rangle \\ 3 \rightarrow 2 : \begin{cases} \text{ok} : G_{\text{sign}} \\ \neg\text{ok} : 3 \rightarrow 1 : \mathbf{abort} \end{cases} \\ \neg\text{ok} : \mathbf{end} \end{cases}$ </td> <td colspan="2"></td> </tr> <tr> <td style="vertical-align: top; padding: 5px;">$G_{\text{sign}} :$</td> <td style="padding: 5px;"> $(s.1) \ 3 \rightarrow 2 : \langle \text{signature}_1 \rangle$ $(s.1) \ 2 \rightarrow 3 : \begin{cases} \text{ok} : 2 \rightarrow 3 : \langle \text{signature}_2 \rangle \\ 3 \rightarrow 1 : \begin{cases} \text{success} : 3 \rightarrow 1 : \langle \text{address} \rangle \\ 1 \rightarrow 3 : \langle \text{date} \rangle \\ \neg\text{success} : 1 \rightarrow 3 : G_{\text{resolve}_1} \end{cases} \\ \neg\text{ok} : 2 \rightarrow 1 : G_{\text{resolve}_2} \end{cases}$ </td> <td colspan="2"></td> </tr> </table>				$G_{\text{init}} :$	$(I.1) \ 3 \rightarrow 1 : \langle \text{Title} \rangle$ $(I.2) \ 1 \rightarrow \{2, 3\} : \langle \text{quote} \rangle$ $(I.3) \ 3 \rightarrow 2 : \langle \text{quote}' \rangle$ $(I.4) \ 2 \rightarrow \{1, 3\} : \begin{cases} \text{ok} : G_{\text{contract}} \\ \neg\text{ok} : \mathbf{end} \end{cases}$	$G_b :$	$(1') \ 1 \rightarrow 2 : \langle T \rangle$ $T = (G_{\text{contract}}) _1$	$G_{\text{contract}} :$	$(c.1) \ 1 \rightarrow \{2, 3\} : \langle \text{contract} \rangle$ $(c.2) \ 3 \rightarrow 2 : \langle \text{promise} \rangle$ $(c.3) \ 2 \rightarrow 3 : \begin{cases} \text{ok} : 2 \rightarrow 3 : \langle \text{promise} \rangle \\ 3 \rightarrow 2 : \begin{cases} \text{ok} : G_{\text{sign}} \\ \neg\text{ok} : 3 \rightarrow 1 : \mathbf{abort} \end{cases} \\ \neg\text{ok} : \mathbf{end} \end{cases}$			$G_{\text{sign}} :$	$(s.1) \ 3 \rightarrow 2 : \langle \text{signature}_1 \rangle$ $(s.1) \ 2 \rightarrow 3 : \begin{cases} \text{ok} : 2 \rightarrow 3 : \langle \text{signature}_2 \rangle \\ 3 \rightarrow 1 : \begin{cases} \text{success} : 3 \rightarrow 1 : \langle \text{address} \rangle \\ 1 \rightarrow 3 : \langle \text{date} \rangle \\ \neg\text{success} : 1 \rightarrow 3 : G_{\text{resolve}_1} \end{cases} \\ \neg\text{ok} : 2 \rightarrow 1 : G_{\text{resolve}_2} \end{cases}$		
$G_{\text{init}} :$	$(I.1) \ 3 \rightarrow 1 : \langle \text{Title} \rangle$ $(I.2) \ 1 \rightarrow \{2, 3\} : \langle \text{quote} \rangle$ $(I.3) \ 3 \rightarrow 2 : \langle \text{quote}' \rangle$ $(I.4) \ 2 \rightarrow \{1, 3\} : \begin{cases} \text{ok} : G_{\text{contract}} \\ \neg\text{ok} : \mathbf{end} \end{cases}$	$G_b :$	$(1') \ 1 \rightarrow 2 : \langle T \rangle$ $T = (G_{\text{contract}}) _1$												
$G_{\text{contract}} :$	$(c.1) \ 1 \rightarrow \{2, 3\} : \langle \text{contract} \rangle$ $(c.2) \ 3 \rightarrow 2 : \langle \text{promise} \rangle$ $(c.3) \ 2 \rightarrow 3 : \begin{cases} \text{ok} : 2 \rightarrow 3 : \langle \text{promise} \rangle \\ 3 \rightarrow 2 : \begin{cases} \text{ok} : G_{\text{sign}} \\ \neg\text{ok} : 3 \rightarrow 1 : \mathbf{abort} \end{cases} \\ \neg\text{ok} : \mathbf{end} \end{cases}$														
$G_{\text{sign}} :$	$(s.1) \ 3 \rightarrow 2 : \langle \text{signature}_1 \rangle$ $(s.1) \ 2 \rightarrow 3 : \begin{cases} \text{ok} : 2 \rightarrow 3 : \langle \text{signature}_2 \rangle \\ 3 \rightarrow 1 : \begin{cases} \text{success} : 3 \rightarrow 1 : \langle \text{address} \rangle \\ 1 \rightarrow 3 : \langle \text{date} \rangle \\ \neg\text{success} : 1 \rightarrow 3 : G_{\text{resolve}_1} \end{cases} \\ \neg\text{ok} : 2 \rightarrow 1 : G_{\text{resolve}_2} \end{cases}$														

Fig. 2. Global Types for the Trusted Buyer-Seller Protocol (Sect. 2).

Typing judgements for expressions and processes are of the form $\Gamma \vdash e : S$ or $\Gamma \vdash P \triangleright \Delta$, where $\Gamma ::= \emptyset \mid \Gamma, x : S$ and $\Delta ::= \emptyset \mid \Delta, c : T$. The *standard environment* Γ assigns variables to sorts and service names to closed global types; the *session environment* Δ associates channels to local types. We write $\Gamma, x : S$ only if $x \notin \text{dom}(\Gamma)$, where $\text{dom}(\Gamma)$ denotes the domain of Γ . We adopt the same convention for $a : G$ and $c : T$, and write Δ, Δ' only if $\text{dom}(\Delta) \cap \text{dom}(\Delta') = \emptyset$. Typing rules are as in [5, 10]; as discussed in those works, typability for \mathbb{S} processes ensure communication correctness in terms of *session fidelity* (well-typed processes respect prescribed local protocols) and *communication safety* (well-typed processes do not feature communication errors), among other properties.

5.2 Satisfiability of Local Formulas from \mathbf{A}

Following the approach in [12], properties of processes in \mathbf{A} will be established via analysis of *traces*, which describe the possible executions of a process. This will allow us to prove communication correctness of \mathbb{S} processes, using encoding $\llbracket \cdot \rrbracket$.

Definition 1 (Traces of P [12]). Given a ground process $P \in \mathbf{A}$, we define the set of traces of P , denoted by $\mathbf{traces}(P)$, as

$$\mathbf{traces}(P) = \left\{ [F_1, \dots, F_n] \mid (\emptyset, \{P\}, \emptyset, \emptyset) \xrightarrow{F_1} \dots \xrightarrow{F_n} (\mathcal{E}_n, \mathcal{S}_n, \mathcal{P}_n, \sigma_n, \mathcal{L}_n) \right\}$$

We will denote by \mathbf{tr}_P , a trace from a set $\mathbf{traces}(P)$, for some process P . We will write \mathbf{tr} when P is clear from the context. Notice that, $\mathbf{tr}_P = \mathbf{tr}_Q$ does not necessarily imply that $P = Q$: each process may implement more than one session in different ways.

SAPIC and Tamarin [14] consider two sorts: \mathbf{temp} and \mathbf{msg} . Each variable of sort \mathbf{s} will be interpreted in the domain $D(\mathbf{s})$; in particular, we will denote by $\mathcal{V}_{\mathbf{temp}}$ the set of temporal variables, which is interpreted in the domain $D(\mathbf{temp}) = \mathcal{Q}$; also, $\mathcal{V}_{\mathbf{msg}}$ is the set of message variables, which is interpreted in the domain $D(\mathbf{msg}) = \mathcal{M}$. Below, we will adopt a function $\theta : \mathcal{V} \rightarrow \mathcal{M} \cup \mathcal{Q}$ that maps variables to terms respecting the variable's sorts, that is $\theta(x : \mathbf{s}) \in D(\mathbf{s})$.

Definition 2 (Trace atoms [12]). A trace atom has of one of the forms:

$$A ::= \perp \mid t_1 \approx t_2 \mid i < j \mid i \doteq k \mid F@i$$

denoting, respectively, false, term equality, timepoint ordering, timepoint equality, or an action for a fact F and a timepoint i . The construction of trace formula φ respects the usual first-order convention:

$$\varphi, \psi ::= A \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid \varphi \leftrightarrow \psi \mid (\exists x : \mathbf{s}).\varphi \mid (\forall x : \mathbf{s}).\varphi$$

Given a process P , in the definition below, \mathbf{tr} denotes a trace in $\mathbf{traces}(P)$, $\mathit{id}x(\mathbf{tr})$ denotes the positions in \mathbf{tr} , and \mathbf{tr}_i denotes the i -th position in \mathbf{tr} .

Definition 3 (Satisfaction relation [12]). The satisfaction relation $(\mathbf{tr}, \theta) \models \varphi$ between a trace \mathbf{tr} , a valuation θ , and a trace formula φ is defined as follows

$$\begin{aligned} (\mathbf{tr}, \theta) \models \perp & \quad \text{never} & (\mathbf{tr}, \theta) \models t_1 \approx t_2 & \quad \text{iff } t_1\theta =_E t_2\theta \\ (\mathbf{tr}, \theta) \models i < j & \quad \text{iff } \theta(i) < \theta(j) & (\mathbf{tr}, \theta) \models \neg\varphi & \quad \text{iff not } (\mathbf{tr}, \theta) \models \varphi \\ (\mathbf{tr}, \theta) \models i \doteq j & \quad \text{iff } \theta(i) = \theta(j) & (\mathbf{tr}, \theta) \models \varphi_1 \wedge \varphi_2 & \quad \text{iff } (\mathbf{tr}, \theta) \models \varphi_1 \text{ and } (\mathbf{tr}, \theta) \models \varphi_2 \\ (\mathbf{tr}, \theta) \models F@i & \quad \text{iff } \theta(i) \in \mathit{id}x(\mathbf{tr}) \text{ and } F\theta =_E \mathbf{tr}_{\theta(i)} \\ (\mathbf{tr}, \theta) \models (\exists x : \mathbf{s}).\varphi & \quad \text{iff there exists } u \in D(\mathbf{s}) \text{ such that } (\mathbf{tr}, \theta[x \mapsto u]) \models \varphi \end{aligned}$$

Satisfaction of $(\forall x : \mathbf{s}).\varphi$, $\varphi \vee \psi$ and $\varphi \Rightarrow \psi$ can be obtained from the cases above.

5.3 From Local Types to Local Formulas

Below we assume s is an established session between participants \mathbf{p} and \mathbf{q} . Given $k : \mathbf{temp}$ and a trace formula φ , we write $\varphi(k)$ to say that there is a fact F such that $F@k$ is an atom in φ . Below we assume that S is a subset of \mathbf{msg} .

Definition 4 (Local Formula). Given a local type T and an endpoint $s[\mathbf{p}]$, its local formula $\Phi_{s[\mathbf{p}]}(T)$ is defined inductively as follows:

$$\begin{aligned}
\Phi_{s[\mathbf{p}]}(!\langle \mathbf{q}, S \rangle.T) &= \exists i, z. (\text{out}(s_{\mathbf{p}\mathbf{q}}, z)@i \wedge \psi(\Phi_{s[\mathbf{p}]}(T))) \\
\Phi_{s[\mathbf{p}]}(? \langle \mathbf{q}, U \rangle.T) &= \exists i, z. (\text{inp}(s_{\mathbf{p}\mathbf{q}}, z)@i \wedge \psi(\Phi_{s[\mathbf{p}]}(T))) \\
\Phi_{s[\mathbf{p}]}(\oplus \langle \mathbf{q}, \{l_i : T_i\}_{i \in I} \rangle) &= \exists i. \bigvee_{j \in I} (\text{sel}(s_{\mathbf{p}\mathbf{q}}, l_j)@i \wedge \psi(\Phi_{s[\mathbf{p}]}(T_j))) \\
\Phi_{s[\mathbf{p}]}(\& \langle \mathbf{q}, \{l_i : T_i\}_{i \in I} \rangle) &= \exists i. \bigvee_{j \in I} (\text{bra}(s_{\mathbf{p}\mathbf{q}}, l_j)@i \wedge \psi(\Phi_{s[\mathbf{p}]}(T_j))) \\
\Phi_{s[\mathbf{p}]}(\text{end}) &= \exists i. \text{close}@i.
\end{aligned}$$

where $\psi(\Phi_{s[\mathbf{p}]}(T)) := \forall k. (\Phi_{s[\mathbf{p}]}(T)(k) \Rightarrow i \leq k)$ the quantified variables have sorts $i, j, k : \mathbf{temp}$ and $z : S$, and variables i and z are fresh. The extension of $\Phi(-)$ to session environments, denoted $\widehat{\Phi}(-)$, is as expected: $\widehat{\Phi}(\Delta, s[\mathbf{p}] : T) = \widehat{\Phi}(\Delta) \wedge \Phi_{s[\mathbf{p}]}(T)$.

Remark 2. Since each local type is associated to a unique local formula, the mapping $\Phi_{s[\mathbf{p}]}$ is invertible. That said, from a local formula φ we can obtain the corresponding type $\Phi^{-1}(\varphi)$. For instance, for the local formula $\varphi_{\text{out}} := \exists i, z. (\text{out}(s_{\mathbf{p}\mathbf{q}}, z)@i \wedge \psi(\Phi_{s[\mathbf{p}]}(T)))$, one has $\Phi^{-1}(\varphi_{\text{out}}) = s[\mathbf{p}] : !\langle \mathbf{q}, S \rangle. \Phi_{s[\mathbf{p}]}^{-1}(\varphi')$. The other cases are similar.

The following theorems give a bi-directional connection between (a) well-typedness and (b) satisfiability of the corresponding local formulas (see [15]):

Theorem 4. Let $\Gamma \vdash P \triangleright \Delta$ be a well-typed S process. Also, let $\text{tr} \in \text{traces}(\llbracket [P] \rrbracket)$. Then there exists a θ such that $(\text{tr}, \theta) \models \widehat{\Phi}(\Delta)$.

Theorem 5. Let tr and φ be a trace and a local formula, respectively. Suppose θ is an instantiation such that $(\text{tr}, \theta) \models \varphi$. Then there is a $P \in S$ such that

$$\Gamma_\varphi \vdash P \triangleright \Phi^{-1}(\varphi) \quad \text{where } \Gamma_\varphi = \{\theta(x) : \text{sort}(x) \mid x \in \text{dom}(\theta)\}$$

Example 2. The projection of G_{init} onto participant 3 (Buyer1), under session s is: $s[3] : !\langle 1, \text{string} \rangle. ? \langle 1, \text{int} \rangle. !\langle 2, \text{int} \rangle. \& \langle 2, \{\text{ok} : (G_{\text{contract}}|_3), \neg \text{ok} : \text{end}\} \rangle$.

The local formula associated is:

$$\begin{aligned}
\Phi_{s[3]}(T) &= \exists i_1, z_1. \text{out}(s_{31}, z_1)@i_1 \wedge (\exists i_2, z_2. \text{inp}(s_{31}, z_2)@i_2 \wedge (\exists i_3, z_3. \text{out}(s_{32}, z_3)@i_3 \\
&\quad \wedge (\exists i_4, i_5, z_4. ((\text{bra}(s_{32}, \text{ok})@i_4 \wedge \Phi_{s[3]}(T')) \vee \text{bra}(s_{32}, \neg \text{ok})@i_4 \wedge \text{close}@i_5))) \\
&\quad \wedge ((i_1 < i_2 < i_3 < i_4 \wedge \psi(\Phi_{s[3]}(T'))) \vee (i_1 < i_2 < i_4 < i_5 \wedge \psi(\Phi_{s[3]}(T'))))
\end{aligned}$$

where T' is the projection of G_{contract} onto participant 3.

6 Revisiting the Two-Buyer Contract Signing Protocol

We recall the motivating example introduced in Sect. 2. Using a combination of constructs from S and A , we first develop a protocol specification which is compiled down to A using our encoding; the resulting A process can be then used

to verify authentication and protocol correctness properties in SAPIC/Tamarin. Figure 2 shows the corresponding global types, and their associated local types (obtained via projection following [10]).

An alternative approach to specification/verification would be as follows. First, specify the protocol using S only, abstracting away from cryptography, and using existing type systems for S to enforce protocol correctness. Then, compile this resulting S specification down to A , where the resulting specification can be enhanced with cryptographic exchanges and authentication properties can be enforced with SAPIC/Tamarin.

6.1 Process Specification

Process specifications for B_i and S are as follows:

$$\begin{aligned}
 B_1 &= \bar{a}[3](y).y[3]!\langle 1, \text{“Title”} \rangle.y[3]?(1, x_1).y[3]!\langle 2, x_1 \text{ div } 2 \rangle.y[3]\&(2, \{\text{ok} : B_1^{sct}, \neg\text{ok} : \mathbf{0}\}) \\
 B_2 &= a[2](y).y[2]?(1, x_2).y[2]?(3, x_3).\text{if } x_2 - x_3 \leq 99 \text{ then } y[2] \oplus \langle \{1, 3\}, \text{ok} \rangle.B_2^{sct} \\
 &\quad \text{else } y[2] \oplus \langle \{1, 3\}, \neg\text{ok} \rangle.\mathbf{0} \\
 S &= a[1](y).y[1]?(3, x_1).y[1]!\langle 2, 3, \text{quote} \rangle.y[1]\&(2, \{\text{ok} : \bar{b}[2](z).y[2]!\langle \{1, y\} \rangle.z[2]?(1, x_4). \\
 &\quad y[1]!\langle 2, \text{date} \rangle.\mathbf{0}, \neg\text{ok} : \mathbf{0}\})
 \end{aligned}$$

where processes B_1^{sct} and B_2^{sct} , which implement the contract signing phase, are as in Tables 9 and 10, respectively. The specification for the trusted authority T is as follows:

$$\begin{aligned}
 &b[1](z).z[1]?(2, t).\nu sk(T); t[3]!\langle \{1, 2\}, pk(sk(T)) \rangle.y[1]?(3, z_2).y[1]?(2, z_3).(\nu s)\text{insert}((s, \text{init})). \\
 &(\nu ct)t[3]!\langle \{1, 2\}, ct \rangle.t[3]\&(\{1, 2\}, \{\text{abort} : P_{Ab}^T, \text{res}_1 : P_{R_1}^T, \text{res}_2 : P_{R_2}^T, \text{success} : z[1]!\langle 2, \text{ok} \rangle.\mathbf{0}\})
 \end{aligned}$$

where processes P_{Ab}^T , $P_{R_1}^T$, and $P_{R_2}^T$ are given in Tables 11 and 12. Process T illustrates how we may combine constructs from S (important to represent, e.g., session establishment on b and delegation from S) and features from A (essential to, e.g., manipulate the memory cell s , which records contract information). Indeed, T uses the A construct `insert` to initialize the cell s and `lookup - as -` to update it. Therefore, the sound and complete encoding proposed in Sect. 4 allows us to specify processes in A , while retaining the high-level constructs from S .

To model the second phase of the protocol, we consider a Private Contract Signature

$$\begin{aligned}
 \Sigma_{\text{pcs}} = \{ &aenc(-, -), senc(-, -), pk(-), sk(-), \text{pcs}, \text{sign}, \text{tsign}, sdec(-), adec(-), \\
 &sconvert, tconvert, pcsver, sverif \}
 \end{aligned}$$

with function symbols for promises and signatures, and for verifying the validity of exchanged messages. As for constructors: `pcs`(x, y, w, z) is the promise of x to y to sign contract z given by w ; `sign`(x, y) is the signature of x in z ; `pk`(x) is the public key of x ; `sk`(x) is the secret key of x ; `aenc`(x, y) is the asymmetric encryption of y using key x ; and `senc`(x, y) is the symmetric encryption of y using

Table 9. B_1^{sct} : B_1 's contract signing processes. $[m]_X$ denotes $\langle m, \text{sign}(sk(x), m) \rangle$

$$\begin{aligned}
B_1^{sct} &= y[3]?(1, z_1). \nu sk(B_1). y[3]!\langle \{1, 2\}, pk(sk(B_1)) \rangle. y[3]?(2, z_3). y[3]?(1, z_4). \\
&\quad y[3]!(2, m_1). y[3]\&(2, \{\text{ok} : P_{conv}^1, \neg\text{ok} : \mathbf{0}\}) \\
P_{conv}^1 &= y[3]?(2, z_5). (\text{if pcsver}(z_3, pk(sk(B_1))), z_1, z_4, z_5) = \text{true} \text{ then } (y[3] \oplus \langle 2, \text{ok} \rangle). \\
&\quad y[3]!\langle 2, S_1 \rangle. y[3]\&(2, \{\text{ok} : P_{sign}^1, \neg\text{ok} : P_{res}^1\}) \text{ else } y[3] \oplus \langle 1, \text{abort} \rangle. P_{abort}^1) \\
P_{sign}^1 &= y[3]?(2, z_6). (\text{if sver}(z_3, z_4, z_6) = \text{true} \text{ then } (y[3] \oplus \langle 1, \text{success} \rangle). \mathbf{0} \text{ else } P_{res}^1) \\
P_{res}^1 &= y[3] \oplus \langle 1, \text{res}_1 \rangle. y[3]!\langle 1, \langle S_1, x_1 \rangle \rangle. y[3]?(1, z_7). \mathbf{0} \\
P_{abort}^1 &= y[3] \oplus \langle 1, \text{abort} \rangle. y[3]!\langle 1, [ct, B_1, B_2, \text{abort}]_{B_1} \rangle. y[3]?(1, z_8). \mathbf{0}
\end{aligned}$$

Table 10. B_2^{sct} : B_2 's contract signing processes.

$$\begin{aligned}
B_2^{sct} &= y[2]?(1, z_1). y[2]?(1, z_2). \nu sk(B_2). y[2]!\langle \{1, 3\}, pk(sk(B_2)) \rangle. y[2]?(3, z_9). y[2]?(3, z_{10}). \\
&\quad (\text{if pcsver}(z_2, pk(sk(B_2))), z_1, z_4, z_{10}) = \text{true} \text{ then } (y[2] \oplus \langle 3, \text{ok} \rangle. y[2]!\langle 3, m_2 \rangle. \\
&\quad y[2]\&(3, \{\text{ok} : P_{sign}^2, \neg\text{ok} : \mathbf{0}\}) \text{ else } y[2] \oplus \langle 3, \neg\text{ok} \rangle. \mathbf{0}) \\
P_{sign}^2 &= y[2]?(3, z_{11}). \text{if sver}(z_2, z_4, z_{11}) = \text{true} \text{ then } y[2] \oplus \langle 3, \text{ok} \rangle. y[2]!\langle 3, S_2 \rangle. \\
&\quad y[2] \oplus \langle 1, \text{success} \rangle. \mathbf{0} \text{ else } y[2] \oplus \langle 3, \neg\text{ok} \rangle. P_{resolve}^2 \\
P_{resolve}^2 &= y[2] \oplus \langle 1, \text{res}_2 \rangle. y[2]!\langle 1, S_2 \rangle. y[2]?(1, z_{12}). \mathbf{0}
\end{aligned}$$

Table 11. P_{Ab}^T : abort process executed by T

$$\begin{aligned}
P_{Ab}^T &= \text{lock } s; y[1]?(3, y_1). \text{if sver}(fst(y_1), snd(y_1)) = \text{true} \text{ then } (\text{lookup } s \text{ as } y_2 \text{ in} \\
&\quad (\text{if } fst(y_2) = \text{init} \text{ then } (\text{insert}((s, [y_1]_T)); y[1]!\langle 3, [y_1]_T \rangle; \text{unlock } s_{pq})) \\
&\quad \text{else } (\text{if } fst(y_2) = \text{abort} \text{ then } y[1]!\langle 3, y_2 \rangle; \text{unlock } s_{pq})) \\
&\quad \text{else if } fst(y_2) = \text{res}_i \text{ then } y[1]!\langle 3, y_2 \rangle; \text{unlock } s_{pq}))
\end{aligned}$$

key x . Destructor $sdec(\cdot)$ (resp. $adec(\cdot)$) enforces symmetric (resp. asymmetric) decryption; the other destructors ($sconvert$, $tconvert$, $pcsver$, $sverif$) are defined from the rules in E_{pcs} :

$$\begin{aligned}
sdec(x, senc(x, y)) &\rightarrow y & tconvert(w, pcs(x, y, pk(w), z)) &\rightarrow \text{sign}(x, z) \\
adec(sk(x), aenc(pk(x), y)) &\rightarrow y & pcsver(pk(x), y, w, z, pcs(x, y, w, z)) &\rightarrow \text{true} \\
sver(pk(x), z, \text{sign}(x, z)) &\rightarrow \text{true} & sconvert(x, pcs(x, y, w, z)) &\rightarrow \text{sign}(x, z)
\end{aligned}$$

Table 13 shows the translation of B_1 in \mathbf{A} , using our encoding. For simplicity, we omit the details related to the session establishment (using NSL), which follow Table 7. Process specifications for B_2 , S , and T in \mathbf{A} can be obtained similarly. As mentioned in Sect. 4, the communication is done via updating session queues s_{ij} , for $i, j = 1, 2, 3$.

Table 12. $P_{\text{res}_1}^T \text{ e } P_{\text{res}_2}^T$: resolve processes executed by T

$$\begin{aligned}
P_{\text{res}_1}^T &= \text{lock } s; y[1]?(3, y_3). \text{if } m'_{11} =_{E_{pcs}} \text{ true then (if } m'_{12} =_{E_{pcs}} \text{ true then} \\
&\quad (\text{lookup } s \text{ as } y_3 \text{ in (if } fst(y_3) = \text{abort then } y[1]!(3, snd(y_3)).\text{unlock } s)) \\
&\quad \text{else (if } fst(y_3) = \text{res}_2 \text{ then } y[1]!(3, snd(y_3)).z[1]!(2, \text{ok}); \text{unlock } s)) \\
&\quad \text{else } y[1]!(3, \text{tconvert}(sk(T), snd(m'_1))).z[1]!(2, \text{ok}).\text{unlock } s)) \\
P_{\text{res}_2}^T &= \text{lock } s; y[1]?(2, w_3); \text{if } m'_{21} =_{E_{pcs}} \text{ true then (if } m'_{22} =_{E_{pcs}} \text{ true then} \\
&\quad (\text{lookup } s \text{ as } w_3 \text{ in (if } fst(w_3) = \text{abort then } y[1]!(2, snd(w_3)); \text{unlock } s)) \\
&\quad \text{else (if } fst(w_3) = \text{res}_1 \text{ then (} y[1]!(2, snd(w_3)); z[1]!(2, \text{ok}); \text{unlock } s)) \\
&\quad \text{else } y[1]!(3, \text{tconvert}(sk(T), fst(m'_2))); \text{insert}((s, \langle \text{res}_2, snd(w_3) \rangle)); \\
&\quad z[1]!(2, \text{ok}); \text{unlock } s))
\end{aligned}$$

Table 13. Translation of B_1 into A .

$$\begin{aligned}
&\nu s; P_{31}; P_{32}; \text{insert}((\widetilde{s}_{ij}, \emptyset)); \text{event init}(\widetilde{s}_{ij}); \text{event sndchann}(pk(ska_{31}), pk(y_1), s); \\
&\quad \text{out}(u_1, s); \text{event sndchann}(pk(ska_{32}), pk(y_2), s); \text{out}(u_2, s); \\
&\quad \text{lock } s_{31}; \text{lookup } s_{31} \text{ as } x_{31} \text{ in} \\
&\quad \quad \text{insert}((s_{31}, x_{31} \cdot \langle \text{msg}, \text{"Title"} \rangle)); \text{event out}(s_{31}, \text{"Title"}); \text{unlock } s_{31}; \\
&\quad \text{lock } s_{13}; \text{lookup } s_{31} \text{ as } x \text{ in} \\
&\quad \quad \text{if } fst(x) = \langle \text{msg}, z \rangle \text{ then insert}((s_{31}, snd(x))); \text{event inp}(s_{31}, z); \text{unlock } s_{13} \\
&\quad \text{lock } s_{32}; \text{lookup } s_{32} \text{ as } x_{32}; \text{in} \\
&\quad \quad \text{insert}((s_{32}, \langle \text{msg}, \text{"quote"} \rangle)); \text{event out}(s_{32}, \text{"quote"}); \text{unlock } s_{32}; \\
&\quad \text{lock } s_{23}; \text{lookup } s_{23} \text{ as } x_{23} \text{ in (if } fst(x_{23}) = \langle \text{lbl}, \text{ok} \rangle \text{ then} \\
&\quad \quad \text{insert}((s_{23}, snd(x_{23}))); \text{event bra}(s_{23}, \text{accept}); \text{unlock } s_{23}; \llbracket B_1^{sct} \rrbracket \\
&\quad \quad \text{else event bra}(s_{23}, \neg \text{ok}); \text{unlock } s_{23}; \mathbf{0})
\end{aligned}$$

6.2 Using SAPIC/Tamarin to Verify Authentication and Local Session Types

We conclude this section by briefly discussing how to use our developments to verify properties associated to authentication and protocol correctness.

Concerning authentication, we can use SAPIC/Tamarin to check the correctness of the authentication phase implemented by NSL. The proof checks that events $\text{honest}(-)$, $\text{sndnonce}(-, -, -)$, $\text{rcvnonce}(-, -, -)$, and $\text{rcvchann}(-, -, -)$ occur in the order specified by the encoding in Table 7. This way, e.g., the following lemma verifies the correctness of the specification of the fragment of NSL authentication with respect to participant B_2 :

lemma B2_NSL_correctness :

exists - trace

(All $pk_{12} \ pk_{1s} \ pk_2 \ pk_s \ \#i \ \#j \ \#k \ \#l$.

$\text{honest}(pk_{12})@i \ \& \ \text{honest}(pk_{1s})@j \ \& \ \text{honest}(pk_2)@k \ \& \ \text{honest}(pk_s)@l$

$\implies (\text{Ex } x \ y \ z \ s \ \#j_1 \ \#k_1 \ \#l_1. \text{rcvnonce}(pk_{12}, pk_2, x, y)@j_1 \ \& \ \text{sndnonce}(pk_2, pk_{12}, z)@k_1$
 $\quad \& \ \text{rcvchann}(pk_{12}, pk_2, s)@l_1 \ \& \ j_1 < k_1 \ \& \ k_1 < l_1))$

The lemma below says that the session channel exchanged using NSL is secret. The proof relies on **asymmetric-encryption**, which is built in the Tamarin library.

```

lemma Chann_is_secret :
  (All pk12 pk2 pk1s pk_s s z n x y w z n2 #i #j #l #i1 #i2 #j1 #j2 #k1 #l1 #l2.
  (honest(pk12)@i & honest(pk2)@j & honest(pk1s)@k & honest(pk_s)@l
    & sndnonce(pk2, pk12, z)@i1 & rcvnonce(pk2, pk12, n, z)@j1 & sndnonce(pk12, pk2, w)@i2
    & rcvnonce(pk12, pk2, n2, w)@j2 & sndnonce(pk_s, pk1s, x)@i3 & rcvnonce(pk_s, pk1s, y, x)@j3
    & sndchann(pk12, k2, s)@k1 & rcvchann(pk12, pk2, s)@k2 & sndchann(pk1s, pk_s, s)@l1
    & rcvchann(pk12, pk2, s)@l2)  $\implies$  not(Ex #j. KU(s)@j))

```

We now consider properties associated to fidelity/safety of processes with respect to their local types. The lemma below ensures protocol fidelity of B_1 and B_2 with respect to the corresponding projections of the global type G_{init} , presented in Fig. 2. The corresponding local formula can be obtained following Definition 4:

```

lemma B1.B2_protocol_fidelity :
exists - trace
  (Ex x y z s #j #j1 #k #k1 #l.out(s31, x)@j & inp(s31, z)@k & out(s32, s)@l & j < k & k < l
    & ((bra(s32, ok))@j1 & l < j1) | (bra(s32, -ok)@k1 & l < k1 &  $\Phi(G_{\text{contract}}|_3)$ ) &
  (Ex x y z s #j #j1 #k #k1 #l.inp(s21, z)@k & inp(s23, s)@l & j < k & k < l
    & ((sel(s23, ok)@j1 & l < j1 &  $\Phi(G_{\text{contract}}|_2)$ ) | (sel(s23, -ok)@k1 & l < k1))

```

Using similar lemmas, we can also prove protocol fidelity for processes S and T with respect to the projections of the global types presented in Fig. 2.

7 Related Works and Concluding Remarks

We have connected two distinct process models: the calculus **S** for multiparty session-based communication [10] and the calculus **A** for the analysis of security protocols [12]. To our knowledge, this is the first integration of sessions (in the sense of [11]) within process languages for security protocol analysis. Indeed, research on security extensions to behavioral types (cf. the survey [2]) seems to have proceeded independently from approaches such as those overviewed in [7]. The work in [6] is similar in spirit to ours, but is different in conception and details, as it uses a session graph specification to generate a cryptographic functional implementation that enjoys session integrity. Extensions of session types (e.g., [4, 16]) address security issues in various ways, but do not directly support cryptographic operations, global state, nor connections with “applied” languages for (automated) verification, which are all enabled by our approach.

Our work should be mutually beneficial for research on (a) behavioral types and contracts and on (b) automated analysis of security protocols: for the former, our work enables the analysis of security properties within multiparty session protocols; for the latter, our approach enables protocol specifications enriched with high-level communication structures based on sessions. In ongoing work, we have used SAPIC/Tamarin to implement our encodings and the verification technique for communication correctness, based on local formulas (Definition 4). Results so far are very promising, as discussed in Sect. 6.

In future work, we intend to explore our approach to process specification and verification in the setting of ProVerif [3], whose input language is a typed applied π -calculus. We also plan to connect our approach with existing type systems for secure information flow and access control in multiparty sessions [4].

Acknowledgements. We are grateful to the anonymous reviewers for their useful remarks and suggestions. Pérez is also affiliated to the NOVA Laboratory for Computer Science and Informatics (supported by FCT grant NOVA LINCSPEst/UID/CEC/04516/2013), Universidade Nova de Lisboa, Portugal.




References

1. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: Proceedings of POPL 2001, pp. 104–115 (2001)
2. Bartoletti, M., Castellani, I., Deniérou, P., Dezani-Ciancaglini, M., Ghilezan, S., Pantovic, J., Pérez, J.A., Thiemann, P., Toninho, B., Vieira, H.T.: Combining behavioural types with security analysis. *J. Log. Algebr. Meth. Program.* **84**(6), 763–780 (2015)
3. Blanchet, B.: Modeling and verifying security protocols with the applied Pi calculus and ProVerif. *Found. Trends Priv. Secur.* **1**(1–2), 1–135 (2016)
4. Capecchi, S., Castellani, I., Dezani-Ciancaglini, M.: Typing access control and secure information flow in sessions. *Inf. Comput.* **238**, 68–105 (2014)
5. Coppo, M., Dezani-Ciancaglini, M., Padovani, L., Yoshida, N.: A gentle introduction to multiparty asynchronous session types. In: Bernardo, M., Johnsen, E.B. (eds.) SFM 2015. LNCS, vol. 9104, pp. 146–178. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18941-3_4
6. Corin, R., Deniérou, P., Fournet, C., Bhargavan, K., Leifer, J.J.: Secure implementations for typed session abstractions. In: Proceedings of CSF 2007, pp. 170–186. IEEE (2007)
7. Cortier, V., Kremer, S.: Formal models and techniques for analyzing security protocols: a tutorial. *Found. Trends Program. Lang.* **1**(3), 151–267 (2014)
8. Garay, J.A., Jakobsson, M., MacKenzie, P.: Abuse-free optimistic contract signing. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 449–466. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_29
9. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0053567>
10. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: POPL, pp. 273–284 (2008)
11. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniérou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. *ACM Comput. Surv.* **49**(1), 3 (2016)
12. Kremer, S., Künnemann, R.: Automated analysis of security protocols with global state. In: Proceedings of SP 2014, pp. 163–178. IEEE (2014)
13. Lowe, G.: Breaking and fixing the needham-schroeder public-key protocol using FDR. *Softw. Concepts Tools* **17**(3), 93–102 (1996)

14. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN prover for the symbolic analysis of security protocols. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 696–701. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_48
15. Nantes, D., Pérez, J.A.: Relating process languages for security and communication correctness (full version). Technical report (2018). <http://www.jperez.nl>
16. Pfenning, F., Caires, L., Toninho, B.: Proof-carrying code in a session-typed process calculus. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 21–36. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_4



A Calculus for Modeling Floating Authorizations

Jovanka Pantović¹ , Ivan Prokić¹ , and Hugo Torres Vieira² 

¹ Faculty of Technical Sciences, University of Novi Sad, Novi Sad, Serbia
prokić@uns.ac.rs

² IMT School for Advanced Studies Lucca, Lucca, Italy

Abstract. Controlling resource usage in distributed systems is a challenging task given the dynamics involved in access granting. Consider, e.g., the setting of floating licenses where access can be granted if the request originates in a licensed domain and if the number of active users is within the license limits. Access granting in such scenarios is given in terms of floating authorizations, addressed in this paper as first class entities of a process calculus model, encompassing the notions of domain, accounting and delegation. We present the operational semantics of the model in two equivalent alternative ways, each informing on the specific nature of authorizations. We also introduce a typing discipline to single out systems that never get stuck due to lacking authorizations, addressing configurations where authorization assignment is not statically prescribed in the system specification.

1 Introduction

Despite the continuous increase of computational resources, their usage will always nevertheless be subject to accessibility and availability. Regardless whether such resources are of hardware or software in nature, they might have finite or virtually infinite capabilities. Physical examples, that can be mapped to finite capabilities directly, include devices such as printers or cell phones, and components of a computing system such as memory or processors. Virtual examples, such as a shared memory cell or a web service, can be more easily seen as having infinite potential but often their availability is also finitely constrained. In general, ensuring proper resource usage is a crucial yet non trivial task, given the highly dynamic nature of access requests and the flexibility necessary to handle such requests, while ensuring secure and efficient system operation.

For security purposes it is crucial to control that access is granted only to authorized users, but also to enforce that access granting is subject to availability. Concrete examples include a wireless access point that has a given access policy and a limited capacity on the number of connected devices, and a software application that is licensed to be used by an institution in a bounded way. Both examples address limited capabilities that are accessible in a shared way

to authorized users. We thus identify a notion of *floating* authorization, adopting the terminology from the licensing realm, where access to resources can be granted to any authorized user in a given *domain* up to the point the capacity is reached. We also identify a notion of implicit granting since users may be granted access directly when using the resource (e.g., running the licensed software).

Considering the licensing setting, we find further examples that inform on the dynamic nature of authorizations. For example, a user deploys a software application on the cloud and provides the license himself (cf. *Bring Your Own License* [6]), potentially losing access to run the application locally given the capacity constraints. We identify here a notion of authorization *delegation*, where the intention to yield or obtain an authorization is explicit. We therefore distill the following dimensions in a floating authorizations model: domain so as to specify where access may be (implicitly) granted; accounting so as to capture capacity; and delegation so as to model (explicit) authorization granting.

In this paper, we present a model that encompasses these dimensions, developed focusing on a process calculus tailored for communication centered systems. In our model the only resources considered are communication channels, so our exploration on authorization control is carried out considering authorizations refer to channels and their usage (for communication) is controlled. Our development builds on the π -calculus [14], which provides the basis to model communicating systems including name passing, and on a model for authorizations [8], from where we adopt the language constructs for authorization domain scope and delegation. We adapt here the interpretation for authorization domains so as to encompass accounting, which thus allows us to model floating authorizations. To the best of our knowledge, ours is the first process calculus model that addresses floating resources (in our case authorizations) as first class entities. Naturally, this does not mean our language cannot be represented in more canonical models, only that we are unaware of existing approaches that offer abstractions that support reasoning on floating resources in a dedicated way.

After presenting the model, we show a typing discipline that ensures systems never incur in authorization errors, i.e., systems that are blocked due to lacking authorizations. Our type analysis addresses systems for which the authorization assignment is not statically given in the system specification, in particular when authorizations for (some) received names are already held by the receiving parties, a notion we call contextual authorizations. We first discuss some examples that informally introduce our model.

1.1 Model Preview

We present our language by resorting to the licensing setting for the sake of a more intuitive reading. First of all, to model authorization domains we consider a scoping construct, and we write $(license)University$ to represent that the *University* domain holds one *license*. This means that one use of *license* within *University* is authorized. In particular, if *University* comprises two students that are simultaneously active, which we dub *Alice* and *Bob*, we may

write $(license)(Alice \mid Bob)$ in which case either *Alice* or *Bob* can be granted the *license* that is floating, but not both of them.

The idea is to support a notion of accounting, so when one of the students uses the license the scope is confined accordingly. For example, if *Bob* uses the license and evolves to *LicensedBob* then the system evolves to $Alice \mid (license)LicensedBob$ where the change in the scope denotes that *license* is not available to *Alice* anymore. The evolution of the system is such that the authorization is directly confined to *Bob*, who is using it, so as to model implicit granting. At this point *Alice* cannot implicitly grab the authorization and gets stuck if she tries to use *license*. Hence, name *license* may be shared between *Alice* and *Bob*, so the (change of) scoping does not mean the name is privately held by *LicensedBob*, just the authorization.

Now consider that *Bob* and *Carol* want to explicitly exchange an authorization, which can be carried out by a delegation mechanism supported by two communication primitives. We write $auth\langle license \rangle.UnlicensedBob$ to represent the explicit delegation of one authorization for *license* via communication channel *auth*, after which activating configuration *UnlicensedBob*. Instead, by $auth\langle license \rangle.LicensedCarol$ we represent the dual primitive that allows to receive one authorization for *license* via channel *auth* after which activating configuration *LicensedCarol*. So by

$$(license)(auth)auth\langle license \rangle.UnlicensedBob \mid (auth)auth\langle license \rangle.LicensedCarol$$

we represent a system where the authorization for *license* can be transferred leading to

$$(auth)UnlicensedBob \mid (auth)\langle license \rangle.LicensedCarol$$

where the scope of the authorization for *license* changes accordingly. The underlying communication is carried out by a synchronization on channel *auth*, for which we remark the respective authorizations (*auth*) are present. In fact, in our model channels are the only resources and their usage is subject to the authorization granting mechanism.

Our model supports a form of fairness and does not allow a “greedy” usage of resources. For example, in the configuration $(license)(Alice \mid (license)LicensedBob)$ the user *LicensedBob* can only be granted the closest (*license*) first, not interfering with *Alice*, but can be also granted the top-level *license* if he needs both licenses.

We remark that no name passing is involved in the authorization delegation mechanism and that name *license* is known to both ends in the first place. Instead, name passing is supported by dedicated primitives, namely

$$(comm)comm!license.Alice \mid (comm)comm?x.Dylan$$

represents a system where the name *license* can be passed via a synchronization on channel *comm*, leading to the activation of *Alice* and *Dylan* where, in the

latter, the placeholder x is instantiated by *license*. Notice that the synchronization can take place since the authorizations to use channel *comm* are given, one for each endpoint.

Name passing allows to model systems where access to channels changes dynamically (since the communicated names refer to channels) but, as hinted in the previous examples, knowing a name does not mean being authorized to use it. So, for instance, $(comm)comm?x.x!reply.0$ specifies a reception in (authorized) *comm* where the received name is then used to output *reply*, leading to an inactive state (denoted by 0). Receiving *license* in channel *comm* leads to $(comm)license!reply.0$ where the authorization for *comm* is still present but no authorization for *license* is acquired as a result of the communication. Hence the output specified using *license* is not authorized and cannot take place. We remark that an authorization for *reply* is not required, hence communicating a name does not entail usage for the purpose of authorizations. By separating name passing and authorization delegation we are then able to model systems where unauthorized intermediaries (e.g., brokers) may be involved in forwarding names between authorized parties without ever being authorized to use such names. For example $(comm)comm?x.(forward)forward!x.0$. which requires no further authorizations.

Configuration $(comm)comm?x.(auth)auth(x).LicensedDylan$ shows a possible pattern for authorizing received names where, after the reception on *comm*, an authorization reception for the received name (using placeholder x) via *auth* is specified, upon which the authorization to use the received name is acquired. Another possibility for enabling authorizations for received names is to use the authorization scoping, e.g., $(comm)comm?x.(x)LicensedDylan$ where the authorization (x) is instantiated by the received name. This example hints on the fact that the authorization scoping is a powerful mechanism that may therefore be reserved in some circumstances to the Trusted Computing Base, resorting to authorization delegation elsewhere.

To introduce the last constructs of our language, consider the system

$$!(license)license?x.(x)license(x).0 \mid (\nu fresh)(license)license!fresh.license(fresh).0$$

where a licensing server is specified on the left hand side, used in the specification given on the right hand side. By $(\nu fresh)Domain$ we represent the creation of a new name which is private to *Domain*, so the specification on the right hand side can be read as first create a name, then send it via channel *license*, after which receive the authorization to use the *fresh* name via channel *license* and then terminate (the received authorization is not actually used in this simple example). On the left hand side, we find a replicated (i.e., repeatably available) reception on channel *license*, after which an authorization scope for the received name is specified that may then be delegated away.

We now present our type analysis returning to the university scenario. Consider $(exam)(minitest)(alice)alice?x.x!task.0$ where there are available authorizations for *exam* and for *minitest*, and where *alice* is waiting to receive the channel on which she will send *task*. Assuming that she can only receive *exam*

or *minitest* the authorizations specified are sufficient. Which authorization will actually be used depends on the received name, so the authorization is implicitly taken when the received channel is used. However, if a name *viva* is sent then the provided authorizations do not suffice.

In order to capture the fact that the above configuration is authorization safe, provided it is inserted in a context that matches the assumptions described previously, our type analysis records the names that can be safely communicated. For instance, we may say that only names *exam* and *minitest* can be communicated in channel *alice*. Also, consider that *exam* and *minitest* can only receive values that are not subject to authorization control. We then denote by $\{alice\}(\{exam, minitest\}(\emptyset))$ the type of channel *alice* in such circumstances, i.e., that it is not subject to replacement (we will return to this point), and that it can be used to communicate *exam* and *minitest* that in turn cannot be used for communication (typed with \emptyset), reading from left to right. Using this information, we can ensure that the specification given for *alice* above is safe, since all names that will possibly be used in communications are contextually authorized.

To analyze the use of the input variable x we then take into account that it can be instantiated by either *exam* or *minitest* (which cannot be used for channel communication) so the type of x is $\{exam, minitest\}(\emptyset)$. Hence the need to talk about possible replacements of a name, allowing us to uniformly address names that are bound in inputs. Our types, denoted $\gamma(T)$, are then built out of two parts, one addressing possible replacements of the name identity itself (γ), and the other informing on the (type of the) names that may be exchanged in the channel (T). The typing assumption $alice : \{alice\}(\{exam, minitest\}(\emptyset))$ informs on the possible contexts where the system above can be safely used. For instance, it is safe to compose with the system $(alice)alice!minitest$ where *minitest* is sent to *alice*, since the name to be sent belongs to the names expected on *alice*. Instead, consider configuration

$$(exam)(minitest)((alice)alice?x.x!task.0 \mid (bob)bob?x.x!task.0)$$

which is also safe and addressed by our typing analysis considering typing assumptions $alice : \{alice\}(\{exam\}(\emptyset))$ and $bob : \{bob\}(\{minitest\}(\emptyset))$. Notice that which authorization is needed by each student is not statically specified in the system, which is safe when both *exam* and *minitest* are sent given the authorization scopes can be confined accordingly. Clearly, the typing specification already yields a specific association.

The typing analysis shown in Sect. 3 addresses such configurations where authorizations for received names may be provided by the context. In Sect. 2 we present the operational semantics of our language considering two equivalent alternatives that inform on the specific nature of authorizations in our model. We refer to the supporting document [13] for additional material, namely proofs of the results reported here.

Table 1. Syntax of processes.

$P ::= 0$	(Inaction)	$a!b.P$	(Output)	$a\langle b \rangle.P$	(Send authorization)
$P \mid P$	(Parallel)	$a?x.P$	(Input)	$a(b).P$	(Receive authorization)
$(\nu a)P$	(Restriction)	$(a)P$	(Authorization)	$!(a)a?x.P$	(Replicated input)

2 A Model of Floating Authorizations

In this section, we present our process model, an extension of the π -calculus [14] with specialized constructs regarding authorizations adopted from a model for authorizations [8]. The syntax of the language is given in Table 1, assuming a countable set of *names* \mathcal{N} , ranged over by $a, b, c, \dots, x, y, z, \dots$. We briefly present the constructs adopted from the π -calculus. An inactive process is represented by 0. By $P \mid P$ we represent two processes simultaneously active, that may interact via synchronization in channels. The name restriction construct $(\nu a)P$ specifies the creation of a channel name a that is known only to the process P . The output prefixed process $a!b.P$ sends the name b on channel a and proceeds as P , and the input prefixed process $a?x.P$ receives a name on channel a and replaces name x in P with the received name.

The remaining constructs involve authorization specifications. Term $(a)P$ is the authorization scoping, representing that process P has one authorization to use channel a . In contrast with name restriction, name a is not private to P . Term $a\langle b \rangle.P$ represents the process that delegates one authorization for name b along name a and proceeds as P . Term $a(b).P$ represents the dual, i.e., a process which receives one authorization for name b along name a and proceeds as $(b)P$. Term $!(a)a?x.P$ allows to specify infinite behavior: the process receives the name along (authorized) name a and substitutes x in P with the received name, which is activated in parallel with the original process. We adopt a simple form of infinite processes, but we remark that a general replication construct can be encoded following standard lines using replicated input, as discussed later.

In $(\nu x)P$, $a?x.P$ and $!(a)a?x.P$ the name x is *binding* with scope P . As usual, we use $\text{fn}(P)$ and $\text{bn}(P)$ to denote the sets of free and bound names in P , respectively. In $(a)P$ occurrence of the name a is free and occurrences of names a and b in processes $a\langle b \rangle.P$ and $a(b).P$ are also free. We remark that in our model authorization scope extrusion is not applicable since a free name is specified, unlike name restriction, and constructs to send and receive authorizations only affect the scope of authorizations. For the rest of presentation we use the following abbreviations: α_a stands for $a!b, a?x, a\langle b \rangle$ or $a(b)$ (including when $b = a$) and $(\nu \bar{a})$ stands for $(\nu a_1) \dots (\nu a_n)$.

2.1 Action Semantics

As in the π -calculus, the essence of the behavior of processes can be seen as communication. Specific to our model is that two processes ready to synchronize on a channel must be authorized to use the channel. For example, consider

that $(a)a!b.P \mid (a)a?x.Q$ can evolve to $(a)P \mid (a)Q\{b/x\}$, since both sending and receiving actions are authorized, while $(a)a!b.P \mid a?x.Q$ lacks the proper authorization on the receiving end, hence the synchronization cannot occur. Another specific aspect of our language is authorization delegation. For example, in $(a)(b)a\langle b\rangle.P \mid (a)a(b).Q$ both actions along name a are authorized and the delegating process has the authorization on b , hence the delegation can take place, leading to $(a)P \mid (a)(b)Q$ where the authorization scope for b changes. If actions along name a are not authorized or the process delegating lacks the authorization for b , then the synchronization is not possible. We formally define the behavior of processes by means of a labeled transition system and afterwards by means of a reduction semantics, which are shown equivalent but inform in somewhat different ways on the specific nature of authorizations in our model.

The labeled transition system (LTS) relies on observable actions, ranged over by α , defined as follows:

$$\alpha ::= (a)^i a!b \mid (a)^i a?b \mid (a)^i (b)^j a\langle b\rangle \mid (a)^i a(b) \mid (\nu b)(a)^i a!b \mid \tau_\omega$$

where ω is of the form $(a)^{i+j}(b)^k$ and $i, j, k \in \{0, 1\}$ and it may be the case that $a = b$. We may recognize the communication action prefixes together with annotations that capture carried/lacking authorizations and bound names. Intuitively, a communication action tagged with $(a)^0$ represents the action lacks an authorization on a , while $(a)^1$ represents the action is carrying an authorization on a . In the case for authorization delegation two such annotations are present, one for each name involved. As in π -calculus, (νb) denotes that the name in the object of the output is bound. In the case of internal steps, the ω identifies the lacking authorizations, and we use τ to abbreviate $\tau_{(a)^0(b)^0}$ where no authorizations are lacking. By $\mathfrak{n}(\alpha)$, $\mathfrak{fn}(\alpha)$ and $\mathfrak{bn}(\alpha)$ we denote the set of all, free and bound names of α .

The transition relation is the least relation included in $\mathcal{P} \times \mathcal{A} \times \mathcal{P}$, where \mathcal{P} is the set of all processes and \mathcal{A} is the set of all actions, that satisfies the rules in Table 2, which we now briefly describe. The rules (L-OUT), (L-IN), (L-OUT-A), (L-IN-A) capture the actions that correspond to the communication prefixes. In each rule the continuation is activated under the scope of the proper authorizations so as to realize *confinement*. The labels are not decorated with any authorizations, representing that the actions are not carrying any authorizations, omitting $(a)^0$ annotations. In contrast, replicated input is authorized by construction, which is why in (L-IN-REP) the label is decorated with the authorization. Rule (L-PAR) lifts the actions of one of the branches (the symmetric rule is omitted) while avoiding unintended name capture; rule (L-RES) says that actions of P are also actions of $(\nu a)P$, provided that the restricted name is not specified in the action; and (L-OPEN) captures the bound output case, opening the scope of the restricted name a , thus allowing for scope extrusion, all three rules adopted from the π -calculus.

Rule (L-SCOPE-INT) shows the case of a synchronization that lacks an authorization on a , so in the conclusion the action exhibited no longer lacks the respective authorization and leads to a state where the authorization scope is no longer

Table 2. LTS rules.

<p>(L-OUT)</p> $a!b.P \xrightarrow{a!b} (a)P$	<p>(L-IN)</p> $a?x.P \xrightarrow{a?b} (a)P\{b/x\}$	<p>(L-OUT-A)</p> $a\langle b \rangle.P \xrightarrow{a\langle b \rangle} (a)P$	<p>(L-IN-A)</p> $a(b).P \xrightarrow{a(b)} (a)(b)P$
<p>(L-IN-REP)</p> $!(a)a?x.P \xrightarrow{(a)a?b} (a)P\{b/x\} \mid !(a)a?x.P$	<p>(L-PAR)</p> $\frac{P \xrightarrow{\alpha} Q \quad \text{bn}(\alpha) \cap \text{fn}(R) = \emptyset}{P \mid R \xrightarrow{\alpha} Q \mid R}$		
<p>(L-RES)</p> $\frac{P \xrightarrow{\alpha} Q \quad a \notin n(\alpha)}{(\nu a)P \xrightarrow{\alpha} (\nu a)Q}$	<p>(L-OPEN)</p> $\frac{P \xrightarrow{(a)^i a!b} Q \quad a \neq b}{(\nu b)P \xrightarrow{(\nu b)(a)^i a!b} Q}$	<p>(L-SCOPE-INT)</p> $\frac{P \xrightarrow{\tau_{\omega(a)}} Q}{(a)P \xrightarrow{\tau_{\omega}} Q}$	<p>(L-SCOPE-EXT)</p> $\frac{P \xrightarrow{\sigma_a} Q}{(a)P \xrightarrow{(a)\sigma_a} Q}$
<p>(L-SCOPE)</p> $\frac{P \xrightarrow{\alpha} Q \quad \tau_{\omega(a)} \neq \alpha \neq \sigma_a}{(a)P \xrightarrow{\alpha} (a)Q}$	<p>(L-COMM)</p> $\frac{P \xrightarrow{(a)^i a!b} P' \quad Q \xrightarrow{(a)^j a?b} Q' \quad \omega = (a)^{2-i-j}}{P \mid Q \xrightarrow{\tau_{\omega}} P' \mid Q'}$		
<p>(L-CLOSE)</p> $\frac{P \xrightarrow{(\nu a)(b)^i b!a} P' \quad Q \xrightarrow{(b)^j b?a} Q' \quad \omega = (b)^{2-i-j} \quad a \notin \text{fn}(Q)}{P \mid Q \xrightarrow{\tau_{\omega}} (\nu a)(P' \mid Q')}$			
<p>(L-AUTH)</p> $\frac{P \xrightarrow{(b)^k (a)^i a\langle b \rangle} P' \quad Q \xrightarrow{(a)^j a(b)} Q' \quad \omega = (a)^{2-i-j} (b)^{1-k}}{P \mid Q \xrightarrow{\tau_{\omega}} P' \mid Q'}$			

present. We use $\omega(a)$ to abbreviate $(a)^2(b)^k$, $(a)^1(b)^k$, and $(b)^{i+j}(a)^1$, for a given b (including the case $b = a$), in which case ω is obtained by the respective exponent decrement. We remark that in contrast to the extrusion of a restricted name via bound output, where the scope floats *up* to the point a synchronization (rule (L-CLOSE) explained below), authorization scopes actually float *down* to the level of communication prefixes (cf. rules (L-OUT), (L-IN), (L-OUT-A), (L-IN-A)), so as to capture confinement. Rule (L-SCOPE-EXT) follows similar lines as (L-SCOPE-INT) as it also refers to lacking authorizations, specifically for the case of an (external) action that is not carrying a necessary authorization. We use a to denote both an action that specifies a as communication subject (cf. α_a) and is annotated with $(a)^0$ (including bound output), and of the form $(b)^i b\langle a \rangle$ where $i \in \{0, 1\}$ (which includes $(a)^1 a\langle a \rangle$). By $(a)_a$ we denote the respective annotation exponent increase. Rule (L-SCOPE) captures the case of an action that is not lacking an authorization on a , in which case the action crosses seamlessly the authorization scope for a .

The synchronization of parallel processes is expressed by the last three rules, omitting the symmetric cases. In rule (L-COMM) one process is able to send and other to receive a name b along name a so the synchronization may take place. If the sending and receiving actions are not carrying the appropriate authorizations, then the transition label τ_{ω} specifies the lacking authorizations (the needed two minus the existing ones). In rule (L-CLOSE) one process is able to send a bound

name a and the other to receive it, along name b , so the synchronization may occur leading to a configuration where the restriction scope is specified (avoiding unintended name capture) so as to finalize the scope extrusion. The authorization delegation is expressed by rule (L-AUTH), where an extra annotation for ω is considered given the required authorization for the delegated authorization. Carried authorization annotations, considered here up to permutation, thus identify, in a compositional way, the requirements for a synchronization to occur.

2.2 Reduction Semantics

Reduction is defined as a binary relation between processes, denoted \rightarrow , so $P \rightarrow Q$ specifies that process P evolves to process Q in one computational step. In order to identify processes which differ syntactically but have the same behavior, we introduce the *structural congruence* relation \equiv , which is the least congruence relation between processes satisfying the rules given in Table 3. Most rules are standard considering structural congruence in the π -calculus. In addition we adopt rules introduced previously [8] that address authorization scopes, including (SC-AUTH-INACT) that allows to discard unused authorizations.

Regarding authorization scoping, we remark there is no rule which relates authorization scoping and parallel composition. Adopting a rule of the sort $(a)(P \mid Q) \equiv (a)P \mid (a)Q$ would represent introducing/discarding one authorization, thus interfering with authorization accounting. We distinguish $(a)(P \mid Q)$ where the authorization is shared between P and Q and $(a)P \mid (a)Q$ where two authorizations are specified, one for each process. Another approach could be a rule of the sort $(a)(P \mid Q) \equiv P \mid (a)Q$, which also may affect the computational power of a process. For example, processes $a!b.0 \mid (a)0$ and $(a)(a!b.0 \mid 0)$ are clearly not to be deemed equal.

Structural congruence is therefore not expressive enough to isolate two authorized processes willing to synchronize. To define the reduction relation we introduce an auxiliary notion of static contexts with one and two holes and operation

Table 3. Structural congruence.

(SC-PAR-INACT) $P \mid 0 \equiv P$	(SC-PAR-COMM) $P \mid Q \equiv Q \mid P$	(SC-PAR-ASSOC) $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
(SC-RES-INACT) $(\nu a)0 \equiv 0$	(SC-RES-SWAP) $(\nu a)(\nu b)P \equiv (\nu b)(\nu a)P$	(SC-REP) $!(a)a?x.P \equiv !(a)a?x.P \mid (a)a?x.P$
(SC-RES-EXTR) $P \mid (\nu a)Q \equiv (\nu a)(P \mid Q) \quad (a \notin \text{fn}(P))$	(SC-ALPHA) $P \equiv_{\alpha} Q \implies P \equiv Q$	
(SC-AUTH-SWAP) $(a)(b)P \equiv (b)(a)P$	(SC-AUTH-INACT) $(a)0 \equiv 0$	(SC-SCOPE-AUTH) $(a)(\nu b)P \equiv (\nu b)(a)P \quad \text{if } a \neq b$

Table 4. Contexts with one and two holes.

$$\mathcal{C}[\cdot] ::= \cdot \mid P \mid \mathcal{C}[\cdot] \mid (a)\mathcal{C}[\cdot] \quad \mathcal{C}[\cdot_1, \cdot_2] ::= \mathcal{C}[\cdot_1] \mid \mathcal{C}[\cdot_2] \mid P \mid \mathcal{C}[\cdot_1, \cdot_2] \mid (a)\mathcal{C}[\cdot_1, \cdot_2]$$

Table 5. *drift* rules.

<p>(C-END)</p> $\text{drift}(\cdot; \emptyset; \tilde{d}) = \cdot$	<p>(C-REM)</p> $\frac{\text{drift}(\mathcal{C}[\cdot]; \tilde{a}; \tilde{d}, c) = \mathcal{C}'[\cdot]}{\text{drift}((c)\mathcal{C}[\cdot]; \tilde{a}, c; \tilde{d}) = \mathcal{C}'[\cdot]}$	<p>(C-SKIP)</p> $\frac{\text{drift}(\mathcal{C}[\cdot]; \tilde{a}; \tilde{d}) = \mathcal{C}'[\cdot] \quad c \notin \tilde{d}}{\text{drift}((c)\mathcal{C}[\cdot]; \tilde{a}; \tilde{d}) = (c)\mathcal{C}'[\cdot]}$
<p>(C-PAR)</p> $\frac{\text{drift}(\mathcal{C}[\cdot]; \tilde{a}; \tilde{d}) = \mathcal{C}'[\cdot]}{\text{drift}(\mathcal{C}[\cdot] \mid R; \tilde{a}; \tilde{d}) = \mathcal{C}'[\cdot] \mid R}$	<p>(C2-SPL)</p> $\frac{\text{drift}(\mathcal{C}_1[\cdot_1]; \tilde{a}; \tilde{d}) = \mathcal{C}'_1[\cdot] \quad \text{drift}(\mathcal{C}_2[\cdot_2]; \tilde{b}; \tilde{e}) = \mathcal{C}'_2[\cdot]}{\text{drift}(\mathcal{C}_1[\cdot_1] \mid \mathcal{C}_2[\cdot_2]; \tilde{a}; \tilde{b}; \tilde{d}; \tilde{e}) = \mathcal{C}'_1[\cdot_1] \mid \mathcal{C}'_2[\cdot_2]}$	
<p>(C2-REM-L)</p> $\frac{\text{drift}(\mathcal{C}[\cdot_1, \cdot_2]; \tilde{a}; \tilde{b}; \tilde{d}, c; \tilde{e}) = \mathcal{C}'[\cdot_1, \cdot_2]}{\text{drift}((c)\mathcal{C}[\cdot_1, \cdot_2]; \tilde{a}, c; \tilde{b}; \tilde{d}; \tilde{e}) = \mathcal{C}'[\cdot_1, \cdot_2]}$	<p>(C2-REM-R)</p> $\frac{\text{drift}(\mathcal{C}[\cdot_1, \cdot_2]; \tilde{a}; \tilde{b}; \tilde{d}; \tilde{e}, c) = \mathcal{C}'[\cdot_1, \cdot_2]}{\text{drift}((c)\mathcal{C}[\cdot_1, \cdot_2]; \tilde{a}; \tilde{b}, c; \tilde{d}; \tilde{e}) = \mathcal{C}'[\cdot_1, \cdot_2]}$	
<p>(C2-SKIP)</p> $\frac{\text{drift}(\mathcal{C}[\cdot_1, \cdot_2]; \tilde{a}; \tilde{b}; \tilde{d}; \tilde{e}) = \mathcal{C}'[\cdot_1, \cdot_2] \quad c \notin \tilde{d}, \tilde{e}}{\text{drift}((c)\mathcal{C}[\cdot_1, \cdot_2]; \tilde{a}; \tilde{b}; \tilde{d}; \tilde{e}) = (c)\mathcal{C}'[\cdot_1, \cdot_2]}$	<p>(C2-PAR)</p> $\frac{\text{drift}(\mathcal{C}[\cdot_1, \cdot_2]; \tilde{a}; \tilde{b}; \tilde{d}; \tilde{e}) = \mathcal{C}'[\cdot_1, \cdot_2]}{\text{drift}(\mathcal{C}[\cdot_1, \cdot_2] \mid R; \tilde{a}; \tilde{b}; \tilde{d}; \tilde{e}) = \mathcal{C}'[\cdot_1, \cdot_2] \mid R}$	

drift that allows to single out configurations where communication can occur. Intuitively, reduction is possible if two processes have active prefixes ready for synchronization and each holds the proper authorization.

Static contexts are defined in Table 4 following standard lines. We use \cdot_1 and \cdot_2 notation to avoid ambiguity, i.e., when $\mathcal{C}[\cdot_1, \cdot_2] = \mathcal{C}'[\cdot_1] \mid \mathcal{C}''[\cdot_2]$ then $\mathcal{C}[P, Q] = \mathcal{C}'[P] \mid \mathcal{C}''[Q]$. Note that in Table 4 there is no case for name restriction construct (*va*), which allows to identify specific names and avoid unintended name capture. Remaining cases specify holes can occur in parallel composition and underneath the authorization scope, the only other contexts underneath which processes are deemed active. We omit the symmetric cases for parallel composition since contexts will be used up to structural congruence.

Operator *drift* plays a double role: on the one hand it is defined only when the hole/holes is/are under the scope of the appropriate number of authorizations in the context; on the other hand, when defined, it yields a context obtained from the original one by removing specific authorizations (so as to capture confinement). In our model, the specific authorizations that are removed for the sake of confinement are the ones nearest to the occurrence of the hole. Operator *drift* is defined inductively on the structure of contexts by the rules shown in Table 5, both for contexts with one hole (rules prefixed with c-) and for contexts with two holes (rules prefixed with c2-). For the one hole case, $\text{drift}(\mathcal{C}[\cdot]; \tilde{a}; \tilde{d})$ takes a context and lists of names \tilde{a} and \tilde{d} , in which the same name can appear more than once. The first list carries the names of authorizations that are to be removed and the second carries the names of authorizations that have already been removed. Similarly, $\text{drift}(\mathcal{C}[\cdot_1, \cdot_2]; \tilde{a}; \tilde{b}; \tilde{d}; \tilde{e})$ takes a context with two holes, two lists of names \tilde{a} and \tilde{b} representing the names of authorizations which are to be removed and two list of names \tilde{d} and \tilde{e} representing names of authorizations already removed. Lists \tilde{a} and \tilde{d} refer to the \cdot_1 hole while \tilde{b} and \tilde{e} refer to the \cdot_2 hole.

We briefly comment on the rules shown in Table 5, reading from conclusion to premises. The rule where the authorization is removed from the context (C-REM) specifies that the name is passed from the “to be removed” list to the “has been removed” list, where we use \tilde{a}, c to refer to the list that contains \tilde{a} and c and likewise for \tilde{d}, c . In the rule where the authorization is preserved in the context (C-SKIP), we check if the name is not in the second list, hence only authorizations that were not already removed proceeding towards the hole can be preserved. This ensures the removed authorizations are the ones nearest to the hole. The rule for parallel composition (C-PAR) is straightforward and the base rule (C-END) is defined only if the first list is empty. This implies that the operator is defined only when all authorizations from the first list are actually removed from the context up to the point the hole is reached. Note that when defining the operator for some context $\mathcal{C}[\cdot]$ and some list of names \tilde{a} that are to be removed from the context, no authorizations have been removed and the respective list \tilde{d} is empty. For example,

$$\begin{aligned}
\text{drift}((a) \cdot ; a; \emptyset) &= \cdot \\
\text{drift}((a)((a) \cdot | R); a; \emptyset) &= (a)(\cdot | R) \\
\text{drift}(\cdot ; a; \emptyset) &\text{ is undefined} \\
\text{drift}((a)(b) \cdot ; a, b; \emptyset) &= \cdot \\
\text{drift}((a) \cdot ; a, b; \emptyset) &\text{ is undefined} \\
\text{drift}((a) \cdot | (b)0; a, b; \emptyset) &\text{ is undefined.}
\end{aligned}$$

Rule (C2-SPL) describes the case for two contexts with one hole each, in which case the respective operator is used to obtain the resulting context, considering the name lists \tilde{a} and \tilde{d} for the context on the left hand side and \tilde{b} and \tilde{e} for the context on the right hand side. The remaining rules follow exactly the same lines of the ones for contexts with one hole, where authorization removal addresses left and right hole in a dedicated way. For example,

$$\begin{aligned}
\text{drift}((b)(a)(a)(\cdot_1 | \cdot_2); a, b; a; \emptyset; \emptyset) &= \cdot_1 | \cdot_2 \\
\text{drift}((b)(a)(\cdot_1 | (a) \cdot_2); a, b; a; \emptyset; \emptyset) &= \cdot_1 | \cdot_2 \\
\text{drift}((a)(b) \cdot_1 | (a) \cdot_2; a, b; a; \emptyset; \emptyset) &= \cdot_1 | \cdot_2 \\
\text{drift}((b)(\cdot_1 | (a)(a) \cdot_2); a, b; a; \emptyset; \emptyset) &\text{ is undefined.}
\end{aligned}$$

The derivation for the case of two holes relies on the derivations for the cases of one hole and is possible only if the axioms for empty contexts hold. Thus, the operator is undefined if the proper authorizations are lacking. As before, lists \tilde{d} and \tilde{e} are used only internally by the operator and we abbreviate $\text{drift}(\mathcal{C}[\cdot_1, \cdot_2]; \tilde{a}; \tilde{b}; \emptyset; \emptyset)$ with $\text{drift}(\mathcal{C}[\cdot_1, \cdot_2]; \tilde{a}; \tilde{b})$.

We may now present the reduction rules, shown in Table 6. Rule (R-COMM) states that two processes can synchronize on name a , passing name b from emitter to receiver, only if both processes are under the scope of, at least one per each process, authorizations for name a . The yielded process considers the context where the two authorizations have been removed by the *drift* operator, and specifies the *confined* authorizations for a which scope only over the continuations of the communication prefixes P and Q . Analogously to (R-COMM), rule (R-AUTH) states that two process can exchange authorization (b) on a name a

Table 6. Reduction rules.

$\frac{\text{(R-COMM)} \quad \text{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a; a) = \mathcal{C}'[\cdot_1, \cdot_2]}{\mathcal{C}[a!b.P, a?x.Q] \rightarrow \mathcal{C}'[(a)P, (a)Q\{b/x\}]}$	$\frac{\text{(R-AUTH)} \quad \text{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a, b; a) = \mathcal{C}'[\cdot_1, \cdot_2]}{\mathcal{C}[a\langle b \rangle.P, a\langle b \rangle.Q] \rightarrow \mathcal{C}'[(a)P, (a)\langle b \rangle Q]}$
$\frac{\text{(R-STRU)} \quad P \equiv P' \rightarrow Q' \equiv Q}{P \rightarrow Q}$	$\frac{\text{(R-NEWC)} \quad P \rightarrow Q}{(\nu a)P \rightarrow (\nu a)Q}$

only if the first process is under the scope of authorization b and if both processes are authorized to perform an action on name a . As before, the yielded process considers the context where the authorizations have been removed by the *drift* operator. The authorization for b is removed for the delegating process and confined to the receiving process so as to model the exchange. Finally, the rule (R-STRU) closes reduction under structural congruence, and rule (R-NEWC) closes reduction under the restriction construct (νa) . Note there are no rules that close reduction under parallel composition and authorization scoping, as these constructs are already addressed by the contexts in (R-COMM) and (R-AUTH). There is also no rule dedicated to replicated input since, thanks to structural congruence rule (SC-REP), a single copy of replicated process may be distinguished and take a part in a synchronization captured by (R-COMM).

As mentioned previously, we may encode a general form of replication, that we may write as $!P$, as

$$(\nu a)((a)a!a.0 \mid !(a)a?x.(P \mid a!a.0))$$

where $a \notin \text{fn}(P)$, since in two steps it reduces to

$$P \mid (\nu a)((a)(P \mid a!a.0) \mid !(a)a?x.(P \mid a!a.0)).$$

where both a copy of P and the original process are active. Notice that since $a \notin \text{fn}(P)$ the process $((a)(P \mid a!a.0))$ does not require any further authorizations on a .

Synchronizations in our model are tightly coupled with the notion of authorization, in the sense that in the absence of the proper authorizations the synchronizations cannot take place. We characterize such undesired configurations, referred to as *error* processes, by identifying the redexes singled-out in the reduction semantics which are stuck due to the lack of the necessary authorizations. This is the case when the premise of the reduction rules does not hold, hence when the *drift* operator is not defined.

Definition 1 (Error). *Process P is an error if $P \equiv (\nu \tilde{c})\mathcal{C}[\alpha_a.Q, \alpha'_a.R]$ and*

1. $\alpha_a = a!b$, $\alpha'_a = a?x$ and $\text{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a; a)$ is undefined, or
2. $\alpha_a = a\langle b \rangle$, $\alpha'_a = a\langle b \rangle$ and $\text{drift}(\mathcal{C}[\cdot_1, \cdot_2]; a, b; a)$ is undefined.

Notice that the definition is aligned with that of reduction, where structural congruence is used to identify a configuration that directly matches one of the redexes given for reduction, but where the respective application of *drift* is undefined.

We may show that (fully authorized) τ transitions match reductions.

Theorem 1 (Harmony). $P \rightarrow Q$ if and only if $P \xrightarrow{\tau} \equiv Q$.

Both presentations of the semantics inform on the particular nature of authorizations in our model. On the one hand, the labeled transition system is more explicit in what concerns authorization manipulation; on the other hand, the reduction semantics identifies the pairs of processes that may synchronize, and is therefore useful to identify communication errors in a direct way and more amenable to use in the proofs of our typing results.

3 Type System

In this section, we present a type discipline that allows to statically identify *safe* processes, hence that do not exhibit unauthorized actions (cf. Definition 1). As mentioned before, our typing analysis addresses configurations where authorizations can be granted contextually. Before presenting the type language, we introduce auxiliary notions that cope with name generation, namely *symbol* annotations and *well-formedness*.

Since the process model includes name restrictions and types contain name identities, we require a symbolic handling of bound names when they are included in type specifications. Without loss of generality, we refine the process model for the purpose of the type analysis adding an explicit symbolic representation of name restrictions. In this way, we avoid a more involved handling of bound names in typing environments.

Formally, we introduce a countable set of symbols \mathcal{S} ranged over by $\mathbf{r}, \mathbf{s}, \mathbf{t}, \dots$, disjoint with the set of names \mathcal{N} , and symbol κ not in $\mathcal{N} \cup \mathcal{S}$. Also, in order to introduce a unique association of restricted names and symbols, we refine the syntax of the name creation construct $(\nu a)P$ in two possible ways, namely $(\nu a : \mathbf{r})P$ and $(\nu a : \kappa)P$, decorated with symbol from \mathcal{S} or with symbol κ , respectively. We use $\text{sym}(P)$ to denote a set of all symbols from \mathcal{S} in process P . Names associated with symbols from \mathcal{S} may be provided contextual authorizations, while names associated with symbol κ may not.

For the purpose of this section we adopt the reduction semantics, adapted here considering refined definitions of structural congruence and reduction. In particular for structural congruence, we omit axiom (SC-RES-INACT) $(\nu a)0 \equiv 0$ and we decorate name restriction accordingly in rules (SC-RES-SWAP), (SC-RES-EXTR) and (SC-SCOPE-AUTH)—e.g., $P \mid (\nu a : \mathbf{r})Q \equiv (\nu a : \mathbf{r})(P \mid Q)$ and $P \mid (\nu a : \kappa)Q \equiv (\nu a : \kappa)(P \mid Q)$ keeping the condition $a \notin \text{fn}(P)$. We remark that the omission of (SC-RES-INACT) is used in process models where name restriction is decorated with typing information (cf. [1]).

Table 7. Typing rules.

(T-STOP)	(T-PAR)	(T-AUTH)
$\Delta \vdash_{\rho} 0$	$\frac{\Delta \vdash_{\rho_1} P_1 \quad \Delta \vdash_{\rho_2} P_2 \quad \text{sym}(P_1) \cap \text{sym}(P_2) = \emptyset}{\Delta \vdash_{\rho_1 \uplus \rho_2} P_1 \mid P_2}$	$\frac{\Delta \vdash_{\rho \uplus \{a\}} P}{\Delta \vdash_{\rho} (a)P}$
(T-NEW)		
$\frac{\Delta, a : \{a\}(T) \vdash_{\rho} P \quad \Delta' = \Delta\{\mathbf{r}/a\} \quad \mathbf{r} \notin \text{sym}(P) \quad a \notin \rho, \text{names}(T)}{\Delta' \vdash_{\rho} (\nu a : \mathbf{r})P}$		
(T-NEW-REP)		
$\frac{\Delta, a : \kappa(T) \vdash_{\rho} P \quad a \notin \rho, \text{names}(T, \Delta)}{\Delta \vdash_{\rho} (\nu a : \kappa)P}$		
(T-OUT)		
$\frac{\Delta \vdash_{\rho} P \quad \Delta(a) = \gamma(\gamma'(T)) \quad \Delta(b) = \gamma''(T) \quad \gamma'' \subseteq \gamma' \quad a \notin \rho \Rightarrow \gamma \subseteq \rho}{\Delta \vdash_{\rho} a!b.P}$		
(T-IN)		
$\frac{\Delta, x : T \vdash_{\rho} P \quad \Delta(a) = \gamma(T) \quad x \notin \rho, \text{names}(\Delta) \quad a \notin \rho \Rightarrow \gamma \subseteq \rho}{\Delta \vdash_{\rho} a?x.P}$		
(T-REP-IN)		
$\frac{\Delta, x : T \vdash_{\{a\}} P \quad \Delta(a) = \gamma(T) \quad x \notin \rho, \text{names}(\Delta) \quad \text{sym}(P) = \emptyset}{\Delta \vdash_{\rho} !(a)a?x.P}$		
(T-DELEG)		
$\frac{\Delta \vdash_{\rho} P \quad \Delta(a) = \gamma(T) \quad a \notin \rho \Rightarrow \gamma \subseteq \rho}{\Delta \vdash_{\rho \uplus \{b\}} a(b).P}$		
(T-RECEP)		
$\frac{\Delta \vdash_{\rho \uplus \{b\}} P \quad \Delta(a) = \gamma(T) \quad a \notin \rho \Rightarrow \gamma \subseteq \rho}{\Delta \vdash_{\rho} a(b).P}$		

The annotations with symbols from \mathcal{S} allow to yield a unique identification of the respective restricted names. The *well-formed* processes we are interested in have unique occurrences of symbols from \mathcal{S} and no occurrences of symbols from \mathcal{S} in the body of replicated inputs. We may show that well-formedness is preserved under (adapted) structural congruence and reduction, and that typable processes are well-formed.

We may now introduce the type language, which syntax is given by $\gamma ::= \varphi \mid \kappa$ and $T ::= \gamma(T) \mid \emptyset$. Types inform on safe instantiations of names that are subject to contextual authorizations, when γ is a set of names from \mathcal{N} and of symbols from \mathcal{S} (denoted φ), and when names are not subject to contextual authorizations (κ). In $\gamma(T)$ type T characterizes the names that can be communicated in the channel, and type \emptyset represents names that cannot be used for communication. A typing environment Δ is a set of typing assumptions of the form $a : T$. We denote by $\text{names}(T)$ the set of names that occur in T and by $\text{names}(\Delta)$ the set of names that occur in all entries of Δ .

The type system is defined by the rules given in Table 7. A typing judgment $\Delta \vdash_\rho P$ states that P uses channels as prescribed by Δ and that P can only be placed in contexts that provide the authorizations given in ρ , which is a multiset of names (from \mathcal{N} , including their multiplicities). The use of a multiset can be motivated by considering process $a!b.0 \mid a?x.0$ that can be typed as $a : \{a\}(\{b\}(\emptyset)) \vdash_\rho a!b.0 \mid a?x.0$ where necessarily ρ contains $\{a, a\}$ specifying that the process can only be placed in contexts that offer two authorizations on name a (one is required per communicating prefix).

We briefly discuss the typing rules. Rule (T-STOP) says that the inactive process is typable using any Δ and ρ . Rule (T-PAR) states that if processes P_1 and P_2 are typed under the same environment Δ , then $P_1 \mid P_2$ is typed under Δ as well. Also if P_1 and P_2 own enough authorizations when contexts provide authorizations ρ_1 and ρ_2 , respectively, then $P_1 \mid P_2$ will have enough authorizations when the context provides the sum of authorizations from ρ_1 and ρ_2 . By $\rho_1 \uplus \rho_2$ we represent the addition operation for multisets which sums the frequencies of the elements. The condition $\text{sym}(P_1) \cap \text{sym}(P_2) = \emptyset$ is necessary to ensure well-formedness. Rule (T-AUTH) says that $(a)P$ and P are typed under the same environment Δ , due to the fact that scoping is a non-binding construct. If P owns enough authorizations when the context provides $\rho \uplus \{a\}$, then $(a)P$ can be safely placed in a context that provides ρ .

Rule (T-NEW) states that if P is typable under an environment that contains an entry for a , then $(\nu a : \mathbf{r})P$ is typed under the environment removing the entry for a and where each occurrence of name a in Δ is substituted by the symbol \mathbf{r} . By $\Delta\{\mathbf{r}/a\}$ we denote the environment obtained by replacing occurrences of a by \mathbf{r} in every assumption in Δ , hence in every type, excluding when a has an entry in Δ . Condition $\mathbf{r} \notin \text{sym}(P)$ is necessary to ensure well-formedness and $a \notin \rho, \text{names}(T)$ says that the context cannot provide authorization for name a and ensures consistency of the typing assumption.

The symbolic representation of a bound name in the typing environment enables us to avoid the case when a restricted (unforgeable) name could be sent to a process that expects to provide a contextual authorization for the received name. For example consider process $(a)(d)a?x.x!c.0 \mid (\nu b : \mathbf{r})(a)a!b.0$ where a contextual authorization for d is specified, a configuration excluded by our type analysis since the assumption for the type of channel a carries a symbol (e.g., $a : \{a\}(\{\mathbf{r}\}(\emptyset))$) for which no contextual authorizations can be provided. Notice that the typing of the process in the scope of the restriction uniformly handles the name, which leaves open the possibility of considering contextual authorizations for the name within the scope of the restriction.

In rule (T-NEW-REP) the difference with respect to rule (T-NEW) is that no substitution is performed, since the environment must already refer to symbol κ in whatever pertains to the restricted name. For example to type process $(\nu b : \kappa)(a)a!b.0$ the type of a must be $\gamma(\kappa(T))$, for some γ and T , where κ identifies the names communicated in a are never subject to contextual authorizations.

Rule (T-OUT) considers that P is typed under an environment where types of names a and b are such that all possible replacements for name b (specified in

γ'') are safe to be communicated along name a (which is formalized by $\gamma'' \subseteq \gamma'$, where γ' is given in the type of a), and also that T (the carried type of b) matches the specification given for a . In such case, the process $a!b.P$ is typed under the same environment. There are two possibilities to ensure name a is authorized, namely the context may provide directly the authorization for name a or it may provide authorizations for all replacements of name a , formalized as $a \notin \rho \Rightarrow \gamma \subseteq \rho$. Notice this latter option is crucial to address contextual authorizations and that in such case γ does not contain symbols, since ρ by definition does not. Rule (T-IN) follows similar principles.

Rule (T-REP-IN) considers the continuation is typed with an assumption for the input bound name x and that the expected context provides authorizations only for name a . In such a case, the replicated input is typable considering the environment obtained by removing the entry for x , which must match the carried type of a , provided that $x \notin \rho, \text{names}(\Delta)$ since it is bound to this process. Also, the fact that process P does not contain any symbol from \mathcal{S} is necessary to ensure the unique association of symbols and names when copies of the replicated process are activated (see the discussion on example (4) at the end of this Section). In that case, process $!(a)a?x.P$ can be placed in any context that conforms with Δ and provides (any) ρ .

Rules (T-DELEG) and (T-RECEP) consider the typing environment is the same in premises and conclusion. The handling of the subject of the communication (a) is similar to, e.g., rule (T-OUT) and the way in which the authorization is addressed in rule (T-RECEP) follows the lines of rule (T-AUTH). In rule (T-DELEG), the authorization for b is added to the ones expected from the context. Notice that in such way no contextual authorizations can be provided for delegation, but the generalization is direct.

For the sake of presenting the results, we say process P is well-typed if $\Delta \vdash_{\emptyset} P$ and Δ only contains assumptions of the form $a : \{a\}(T)$ or $a : \kappa(T)$. At top level the typing assumptions address the free names of the process, which are not subject to instantiation. Free names are either characterized by $a : \{a\}(T)$ which says that a is the (final) instantiation, or by $a : \kappa(T)$ which says that a cannot be granted contextual authorizations. For example, process $(a)a!b.0 \mid (a)(b)a?x.x!c.0$ is typable under the assumption that name b has type $\{b\}(\{c\}(\emptyset))$, while it is not typable under the assumption $\kappa(\{c\}(\emptyset))$. The fact that no authorizations are provided by the context ($\rho = \emptyset$) means that the process P is self-sufficient in terms of authorizations.

We may now present our results, starting by mentioning some fundamental properties. We may show that typing derivations enjoy standard properties (Weakening and Strengthening) and that typing is preserved under structural congruence (Subject Congruence). As usual, to prove typing is preserved under reduction we may also show an auxiliary result that addresses name substitution (cf. Lemma 3.1 [13]). Noticeably, even though subtyping is not present, the result uses an inclusion principle that already hints on substitutability. Our first main result says that typing is preserved under reduction.

Theorem 2 (Subject Reduction). *If P is well-typed, $\Delta \vdash_{\emptyset} P$ and $P \rightarrow Q$ then $\Delta \vdash_{\emptyset} Q$.*

Not surprisingly, since errors involve redexes, the proof of Theorem 2 is intertwined with the proof of the error absence property, given in our second main result which captures the soundness of our typing analysis.

Proposition 1 (Type Soundness). *If P is well-typed then P is not an error.*

As usual, the combination of Theorem 2 and Proposition 1 yields type safety, stating that any configuration reachable from a well-typed one is not an error. Hence type safety ensures that well-typed processes will never incur in a configuration where the necessary authorizations to carry out the communications are lacking.

We extend the example shown in the Introduction to illustrate the typing rules:

$$(alice)alice!exam.0 \mid (exam)(minitest)(alice)alice?x.x!task.0 \quad (1)$$

Considering assumption $alice : \{alice\}(\{exam, minitest\}(\emptyset))$, and assuming that the type of $exam$ is $\{exam\}(\emptyset)$, by rule (T-OUT) we conclude that it is safe to send name $exam$ along $alice$, since the (only) instantiation of $exam$ is contained in the carried type of $alice$. Now let us place process (1) in a context restricting $exam$:

$$(\nu exam : \mathbf{r})((alice)alice!exam.0 \mid (exam)(minitest)(alice)alice?x.x!task.0) \quad (2)$$

To type this process, the assumption for $alice$ specifies type $\{alice\}(\{\mathbf{r}, minitest\}(\emptyset))$, representing that in $alice$ a restricted name can be communicated. Hence, the process shown in (2) cannot be composed with others that rely on contextual authorizations for names exchanged in $alice$, and can only be composed with processes like $(alice)alice?x.(x)x!task$ or that use authorization delegation. Now consider process:

$$!(license)license?x.(\nu exam : \mathbf{r})((x)x!exam.0 \mid (x)(exam)x?y.y!task.0) \quad (3)$$

that models a server that receives a name and afterwards is capable of both receiving (on the lhs) and sending (a fresh name, on the rhs) along the received name. Our typing analysis excludes this process since it specifies a symbol (\mathbf{r}) in the body of a replicated input. In fact, the process may incur in an error, as receiving $alice$ twice leads to:

$$\begin{aligned} &(\nu exam_1 : \mathbf{r})((alice)alice!exam_1.0 \mid (exam_1)(alice)alice?y.y!task.0) \\ &\mid (\nu exam_2 : \mathbf{r})((alice)alice!exam_2.0 \mid (exam_2)(alice)alice?y.y!task.0). \end{aligned} \quad (4)$$

where two copies of the replicated process are active in parallel, and where two different restricted names can be sent on $alice$, hence the error is reached when the contextual authorization does not match the received name (e.g., $(exam_2)(alice)exam_1!task.0$).

In order to address name generation within replicated input, we use distinguished symbol κ that types channels that are never subject to contextual authorizations, even within the restriction scope. Hence replacing the \mathbf{r} annotation by κ in the process shown in (3) does not yield it typable, since a contextual authorization is expected for name *exam* and may lead to an error like before. However, process:

$$!(license)license?x.(\nu exam : \kappa)(alice)alice!exam.0 \quad (5)$$

is typable, hence may be composed with contexts compliant with *alice* : $\{alice\}(\kappa(T))$, i.e., that do not rely on contextual authorizations for names received on *alice*.

4 Concluding Remarks

In the literature, we find a plethora of techniques that address resource usage control, ranging from locks that guarantee mutual exclusion in critical code blocks to communication protocols (e.g., token ring). Several typing disciplines have been developed to ensure proper resource usage, such as [5, 10, 15], where capabilities are specified in the type language, not as a first class entity in the model. Therefore in such approaches it is not possible to separate resource and capability like we do. We distinguish an approach that considers accounting [5], in the sense that the types specify the number of messages that may be exchanged, therefore related to the accounting presented here.

We also find proposals of models that include capabilities as first class entities, addressing usage of channels and of resources as communication objects, such as [4, 9, 12, 16]. More specifically, constructs for restricting (hiding and filtering) the behaviors allowed on channels [9, 16], usage specification in a (binding) name scope construct [12], and authorization scopes for resources based on given access policies [4]. We distinguish our approach from [9, 16] since the proposed constructs are static and are not able to capture our notion of a floating resource capability. As for [12], the usage specification directly embedded in the model resembles a type and is given in a binding scoping construct, which contrasts with our non-binding authorization scoping. Also in [4] the provided detailed usage policies are associated to the authorization scopes for resources. In both [4, 12] the models seem less adequate to capture our notion of floating authorizations as access is granted explicitly and controlled via the usage/policy specification, and for instance our notion of confinement does not seem to be directly representable.

We have presented a model of floating authorizations, a notion we believe is unexplored in the existing literature. We based our development on previous work [8] where a certain form of inconsistency when handling the authorization granting in delegation was identified, while in this work granting is handled consistently thanks to the interpretation of accounting. We remark that adding choice to this work can be carried out in expected lines. More interesting would be to consider non-consumptive authorizations, that return to their

original scope after (complete) use. We also presented a typing analysis that addresses contextual authorizations, which we also believe is unexplored in the literature in the form we present it here. Our typing rules induce a decidable type-checking procedure, since rules are syntax directed, provided as usual that a (carried) type annotation is added to name restrictions. Albeit the work presented here is clearly of a theoretical nature, we hope the principles developed here may be conveyed to, e.g., the licensing domain where we have identified related patents [2, 3, 6] for the purpose of certifying license usage.

A notion of substitutability naturally arises in our typing analysis and we leave to future work a detailed investigation of a subtyping relation that captures such notion. We believe our approach can be extended by considering some form of usage specifications like the ones mentioned above [4, 12], by associating to each authorization scoping more precise capabilities in the form of behavioral types [11]. This would also allow us to generalize our approach addressing certain forms of infinite behavior, namely considering recursion together with linearity constraints that ensure race freedom. It would also be interesting to resort to refinement types [7] to carry out our typing analysis, given that our types can be seen to some extent as refinements on the domain of names.

Acknowledgments. We thank anonymous reviewers for useful remarks and suggestions. This work has been partially supported by the Ministry of Education and Science of the Republic of Serbia, project ON174026, and EU COST Action IC1405 (Reversible Computation).



References

1. Acciai, L., Boreale, M.: Spatial and behavioral types in the pi-calculus. *Inf. Comput.* **208**(10), 1118–1153 (2010). <https://doi.org/10.1016/j.ic.2009.10.011>
2. Armstrong, W.J., Nayar, N., Stamschror, K.P.: Management of a concurrent use license in a logically-partitioned computer. US Patent 6,959,291 (2005)
3. Baratti, P., Squartini, P.: License management system. US Patent 6,574,612 (2003)
4. Bodei, C., Dinh, V.D., Ferrari, G.L.: Checking global usage of resources handled with local policies. *Sci. Comput. Program.* **133**, 20–50 (2017). <https://doi.org/10.1016/j.scico.2016.06.005>
5. Das, A., Hoffmann, J., Pfenning, F.: Work analysis with resource-aware session-types. *CoRR* abs/1712.08310 (2017). <http://arxiv.org/abs/1712.08310>
6. Ferris, J.M., Riveros, G.E.: Offering additional license terms during conversion of standard software licenses for use in cloud computing environments. US Patent 9,053,472 (2015)
7. Freeman, T.S., Pfenning, F.: Refinement types for ML. In: Wise, D.S. (ed.) *Proceedings of the PLDI 1991*, pp. 268–277. ACM (1991). <https://doi.org/10.1145/113445.113468>
8. Ghilezan, S., Jakšić, S., Pantović, J., Pérez, J.A., Vieira, H.T.: Dynamic role authorization in multiparty conversations. *Formal Asp. Comput.* **28**(4), 643–667 (2016). <https://doi.org/10.1007/s00165-016-0363-5>
9. Giunti, M., Palamidessi, C., Valencia, F.D.: Hide and new in the pi-calculus. In: *Proceedings of EXPRESS/SOS 2012, EPTCS*, vol. 89, pp. 65–79 (2012). <https://doi.org/10.4204/EPTCS.89.6>

10. Gorla, D., Pugliese, R.: Dynamic management of capabilities in a network aware coordination language. *J. Log. Algebr. Program.* **78**(8), 665–689 (2009). <https://doi.org/10.1016/j.jlap.2008.12.001>
11. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniérou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. *ACM Comput. Surv.* **49**(1), 3:1–3:36 (2016). <https://doi.org/10.1145/2873052>
12. Kobayashi, N., Suenaga, K., Wischik, L.: Resource usage analysis for the p-calculus. *Log. Methods Comput. Sci.* **2**(3) (2006). [https://doi.org/10.2168/LMCS-2\(3:4\)2006](https://doi.org/10.2168/LMCS-2(3:4)2006)
13. Pantovic, J., Prokic, I., Vieira, H.T.: A calculus for modeling floating authorizations. *CoRR abs/1802.05863* (2018). <https://arxiv.org/abs/1802.05863>
14. Sangiorgi, D., Walker, D.: *The Pi-Calculus - A Theory of Mobile Processes*. Cambridge University Press, Cambridge (2001)
15. Swamy, N., Chen, J., Chugh, R.: Enforcing stateful authorization and information flow policies in FINE. In: Gordon, A.D. (ed.) *ESOP 2010*. LNCS, vol. 6012, pp. 529–549. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_28
16. Vivas, J., Yoshida, N.: Dynamic channel screening in the higher order pi-calculus. *Electr. Notes Theor. Comput. Sci.* **66**(3), 170–184 (2002). [https://doi.org/10.1016/S1571-0661\(04\)80421-3](https://doi.org/10.1016/S1571-0661(04)80421-3)



Parameter Synthesis Algorithms for Parametric Interval Markov Chains

Laure Petrucci¹ and Jaco van de Pol²

¹ LIPN, CNRS UMR 7030, Université Paris 13, Sorbonne Paris Cité,
Villetaneuse, France

`laure.petrucci@lipn.univ-paris13.fr`

² Formal Methods and Tools, University of Twente, Enschede, The Netherlands
`J.C.vandePol@utwente.nl`

Abstract. This paper considers the consistency problem for Parametric Interval Markov Chains. In particular, we introduce a co-inductive definition of consistency, which improves and simplifies previous inductive definitions considerably. The equivalence of the inductive and co-inductive definitions has been formally proved in the interactive theorem prover PVS.

These definitions lead to forward and backward algorithms, respectively, for synthesizing an expression for all parameters for which a given PIMC is consistent. We give new complexity results when tackling the consistency problem for IMCs (i.e. without parameters). We provide a sharper upper bound, based on the longest simple path in the IMC. The algorithms are also optimized, using different techniques (dynamic programming cache, polyhedra representation, etc.). They are evaluated on a prototype implementation. For parameter synthesis, we use Constraint Logic Programming and the PARMA library for convex polyhedra.

1 Introduction

Markov Chains (MC) are widely used to model stochastic systems, like randomized protocols, failure and risk analysis, and phenomena in molecular biology. Here we focus on discrete time MCs, where transitions between states are governed by a state probability distribution, denoted by $\mu : S \times S \rightarrow [0, 1]$. Practical applications are often hindered by the fact that the probabilities $\mu(s, t)$, to go from state s to t , are unknown. Several solutions have been proposed, for instance Parametric Markov Chains (PMC) [7] and Interval Markov Chains (IMC) [14], in which unknown probabilities are replaced by parameters or intervals, respectively, see Figs. 1a and b. Following [9], we study their common generalization, Parametric Interval Markov Chains (PIMC, Fig. 1c), which allow intervals with parametric bounds. PIMCs are more expressive than IMCs and PMCs [2]. PIMCs allow to study the boundaries of admissible probability intervals, which is useful in the design exploration phase. This leads to the study of parameter synthesis for PIMCs, started in [12].

This research was conducted with the support of PHC Van Gogh project PAMPAS.

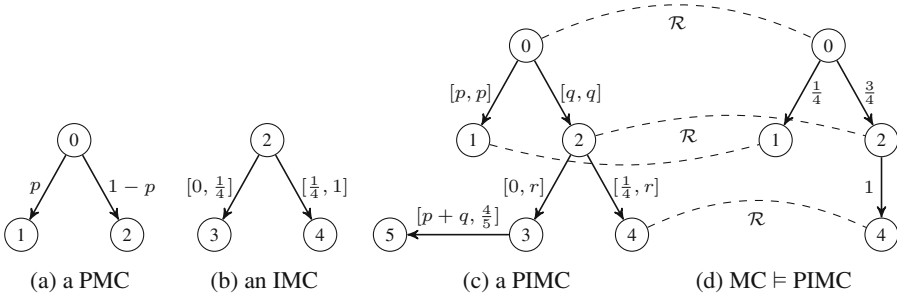


Fig. 1. Examples of a PMC (a), an IMC (b), and of their generalization PIMC (c). The MC (d) implements the PIMC (c), as shown by the dashed edges, and formalized in Definition 7. We drop self-loops with probability 1 (or $[1, 1]$) in all terminal nodes.

IMCs can be viewed as specifications of MCs. An IMC is consistent if there exists an MC that implements it. The main requirements on the implementation are: (1) all behaviour of the MC can be simulated by the IMC, preserving probabilistic information; and (2) the outgoing transition probabilities for each state sum up to 1. The consistency synthesis problem for PIMCs is to compute all parameter values leading to a consistent IMC. E.g., PIMC in Fig. 1c¹ is consistent when $q = 0, p = 1$, or when $p + q = 1, r = 1$. The witness MC of Fig. 1d corresponds to $p = \frac{1}{4}, q = \frac{3}{4}, r = 1$.

Contribution. This paper studies the consistency synthesis problem for PIMCs. We improve the theory in [12], which gave an inductive definition of n -consistency. Basically, a state is n -consistent if it has a locally consistent subset of successors, which are in turn all $(n - 1)$ -consistent. Here local consistency checks that there is a solution within the specified bounds that sums up to 1. That paper provided an expression for n -consistency. There are two drawbacks from an algorithmic perspective: all possible subsets of successors must be enumerated, and a sufficiently large upper bound for n must be provided. It has been shown that taking the number of states for n is sufficient.

We address both problems. First we simplify the inductive definition and show that for IMCs, the enumeration over the subset of successors is not necessary. Instead, we can restrict to a single candidate: the set of *all* $(n - 1)$ -consistent successors. Second, we provide a smaller upper bound for n . We show that it is sufficient to take the length of the longest simple path in the IMC. However, the length of the longest simple path cannot be efficiently computed (this would solve the Hamiltonian path problem, which is NP-complete).

Our main contribution is to provide an alternative, co-inductive definition of consistency, dropping the need to reason about n altogether. Instead, we define consistency as the largest set, such that a state is consistent if the set of its con-

¹ In the following, for the sake of readability, we do not consider linear combinations of parameters as bounds of the intervals. However, allowing them would not change the results.

sistent successors is locally consistent. We have formally proved the equivalence between all these definitions in the interactive theorem prover PVS [17]. The complete PVS proof development is available online ².

Based on the simplified inductive definition, we provide a polynomial time *forward* algorithm to check that an IMC is consistent. Based on the new co-inductive definition, we provide a polynomial *backward* algorithm. Again, the number of iterations is bounded by the length of the longest simple path, without having to compute it.

Finally, we provide algorithms to compute an expression for *all* parameters for which a PIMC is consistent. Unfortunately, to obtain an expression we must fall back on subset enumeration. The forward algorithm can be implemented as a Constraint Logic Program, so Prolog + CLP(Q) can be used directly to compute a list of all solutions, basically as a disjunction of conjunctions of linear inequalities over the parameters. We introduce two optimizations: caching intermediate results and suppressing subsumed solutions. The backward algorithm for IMCs can be viewed as computing the maximal solution of a Boolean Equation System. Generalizing this to PIMCs, we now compute the maximal solution of an equation system over disjunctions of conjunctions of constraints. Such equation systems can be solved by standard iteration, representing the intermediate solutions as powerdomains over convex closed polyhedra. We implemented this using the Parma Polyhedra Library [1].

Related Work. One of the first results on synthesis for PMCs [7] computes the probability of path formulas in PCTL as an expression over the parameters. Since then, the efficiency and numeric stability has been improved considerably [8, 18]. On such models, the realizability (or well-defined) property is considered [13, 16] which mainly differs from the consistency we address in that they consider consistency of all states, while we have the option to avoid some states (and their successors) by assigning null-probability to some edges. Model checking for IMCs is studied, for instance in [5, 6]. For *continuous-time* Markov Chains, precise parameter synthesis is studied as well [4]. However, PIMCs are more expressive (concise) than PMCs and IMCs, as shown in [2]. As far as we know, besides reachability, there are no model checking results for PIMCs.

Other specification formalisms include Constraint Markov Chains [11], with arbitrary constraints on transitions, and Abstract Probabilistic Automata [10], which add non-deterministic transitions. We believe that our work can be extended in a straightforward manner to CMCs with linear constraints. For APA the situation is probably quite different. Another branch of research has investigated the synthesis of parameters for timed systems, but that is out of the scope of this paper.

PIMCs, and the related consistency problem, have been introduced in [9]. Our backward algorithm for IMCs is somewhat similar in spirit to the pruning operator of [9]. We provide a sharper upper bound on the number of required iterations. That paper addresses the *existence* of a consistent parameter valuation

² The complete text of the proofs, their PVS formalisation, Prolog programs, and experimental data can be found at <http://fmt.cs.utwente.nl/~vdpol/PIMC2018.zip>.

for a restricted subclass of PIMCs, where parameters occur only locally. The parameter *synthesis* problem for the full class of PIMCs was considered in [12]. We improved on their theory, as explained before. Our experiments (Sect. 6) show that our algorithms and optimizations are more efficient than the approach in [12].

Very recently, [2] also introduced a CLP approach for checking the *existence* of a consistent parameter valuation. Their contribution is a CLP program of linear size in the PIMC. Their CLP is quite different from ours: basically, they introduce a Boolean variable for each state and a real variable for each transition probability of the Markov Chain that implements the PIMC. So solving the CLP corresponds to searching for a satisfying implementation.

2 Parametric Interval Markov Chains

As Parametric Interval Markov Chains allow for describing a family of Markov Chains, we first define these.

Definition 1 (Markov Chain). A Markov Chain (MC) is a tuple (S, s_0, μ, A, V) , where:

- S is a set of states and $s_0 \in S$ is the initial state;
- $\mu : S \times S \rightarrow [0, 1]$ is the transition probability distribution s.t.:
 $\forall s \in S : \sum_{s' \in S} \mu(s, s') = 1$;
- A is a set of labels and $V : S \rightarrow A$ is the labelling function.

Notation 2. Let P be a set of *parameters*, i.e. variable names. We denote by $\text{Int}[0, 1](P)$ the set of pairs $[a, b]$ with $a, b \in [0, 1] \cup P$. Given $x \in \text{Int}[0, 1](P)$, we denote by x_ℓ and x_u its left and right components. If x is an interval, this corresponds to its lower and upper bounds. The same notation is used for functions which result in an interval.

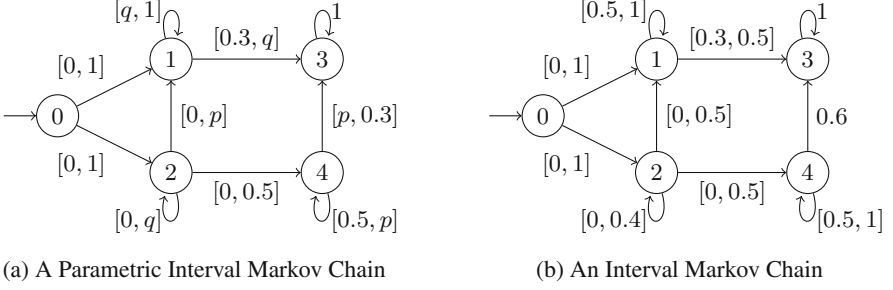
Example 3. $[0.3, 0.7]$, $[0, 1]$, $[0.5, 0.5]$, $[p, 0.8]$, $[0.99, q]$, $[p, q]$ are all in $\text{Int}[0, 1](\{p, q\})$.

Definition 4 ((Parametric) Interval Markov Chain). A Parametric Interval Markov Chain (PIMC) is a tuple $(P, S, s_0, \varphi, A, V)$ where:

- P is a set of parameters;
- S is a set of states and $s_0 \in S$ is the initial state;
- $\varphi : S \times S \rightarrow \text{Int}[0, 1](P)$ is the parametric transition probability constraint;
- A is a set of labels and $V : S \rightarrow A$ is the labelling function.

An Interval Markov Chain (IMC) is a PIMC with $P = \emptyset$ (we drop P everywhere).

Note that Definitions 1 and 4 are very similar, but for PIMCs and IMCs the well-formedness of the intervals and of the probability distribution will be part of the consistency property to be checked (see Definition 11).


Fig. 2. Running examples

When ambiguity is possible, we will use a subscript to distinguish the models, e.g. $S_{\mathcal{M}}$, $S_{\mathcal{I}}$ and $S_{\mathcal{P}}$ will denote the set of states of respectively a MC, an IMC and a PIMC.

If all intervals are point intervals of the form $[p, p]$, this PIMC is actually a Parametric Markov Chain [7].

Example 5. Figure 2a shows our running example of a PIMC with two parameters p and q (taken from [12]). Figure 2b shows a particular IMC.

Definition 6 (Support). *The support of a probability distribution μ at a state $s \in S_{\mathcal{M}}$ is the set: $\text{sup}(\mu, s) := \{s' \in S_{\mathcal{M}} \mid \mu(s, s') > 0\}$.*

Similarly, for a parametric transition probability constraint φ at a state $s \in S_{\mathcal{P}}$ the support is the set: $\text{sup}(\varphi, s) := \{s' \in S_{\mathcal{P}} \mid \varphi_u(s, s') > 0\}$.

Assumption: From now on, we will assume that \mathcal{I} is finitely branching, i.e. for all $s \in S$, $\text{sup}(\varphi, s)$ is a finite set. For the algorithms in Sect. 4 we will even assume that S is finite.

PIMCs and IMCs can be viewed as specifications of MCs.

Definition 7 (A MC implements an IMC). *Let $\mathcal{M} = (S_{\mathcal{M}}, s_{\mathcal{M}0}, \mu, A, V_{\mathcal{M}})$ be a MC and $\mathcal{I} = (S_{\mathcal{I}}, s_{\mathcal{I}0}, \varphi, A, V_{\mathcal{I}})$ an IMC. \mathcal{M} implements \mathcal{I} ($\mathcal{M} \models \mathcal{I}$) if there exists a simulation relation $\mathcal{R} \subseteq S_{\mathcal{M}} \times S_{\mathcal{I}}$, s.t. $\forall s_{\mathcal{M}} \in S_{\mathcal{M}}$ and $s_{\mathcal{I}} \in S_{\mathcal{I}}$, if $s_{\mathcal{M}} \mathcal{R} s_{\mathcal{I}}$, then:*

1. $V_{\mathcal{M}}(s_{\mathcal{M}}) = V_{\mathcal{I}}(s_{\mathcal{I}})$
(the source and target states have the same label)
2. There exists a probabilistic correspondence $\delta : S_{\mathcal{M}} \times S_{\mathcal{I}} \rightarrow [0, 1]$, s.t.:
 - (a) $\forall s'_{\mathcal{I}} \in S_{\mathcal{I}} : \sum_{s'_{\mathcal{M}} \in S_{\mathcal{M}}} \mu(s_{\mathcal{M}}, s'_{\mathcal{M}}) \cdot \delta(s'_{\mathcal{M}}, s'_{\mathcal{I}}) \in \varphi(s_{\mathcal{I}}, s'_{\mathcal{I}})$
(the total contribution of implementing transitions satisfies the specification)
 - (b) $\forall s'_{\mathcal{M}} \in S_{\mathcal{M}} : \mu(s_{\mathcal{M}}, s'_{\mathcal{M}}) > 0 \Rightarrow \sum_{s'_{\mathcal{I}} \in S_{\mathcal{I}}} \delta(s'_{\mathcal{M}}, s'_{\mathcal{I}}) = 1$
(the implementing transitions yield a probability distribution)
 - (c) $\forall s'_{\mathcal{M}} \in S_{\mathcal{M}}, s'_{\mathcal{I}} \in S_{\mathcal{I}} : \delta(s'_{\mathcal{M}}, s'_{\mathcal{I}}) > 0 \Rightarrow s'_{\mathcal{M}} \mathcal{R} s'_{\mathcal{I}}$
(corresponding successors are in the simulation relation)

$$(d) \forall s'_M \in S_M, s'_I \in S_I : \delta(s'_M, s'_I) > 0 \Rightarrow \mu(s_M, s'_M) > 0 \wedge \varphi_u(s_I, s'_I) > 0$$

(δ is only defined on the support of μ and φ)

Definition 8 (Consistency). An IMC \mathcal{I} is consistent if for some MC \mathcal{M} , we have $\mathcal{M} \models \mathcal{I}$.

A PIMC is consistent if there exist parameter values such that the corresponding IMC is consistent.

Intuitively, this definition states that the implementation is an MC, whose behaviour is allowed by the specification IMC, i.e., the IMC can simulate the MC. Clause (2d) was not present in the original definition [9], but it is convenient in proofs. We first show that limiting δ to the support of μ and φ does not alter the implementation relation.

Lemma 9. Definition 7 with clauses (2a)–(2c) is equivalent to Definition 7 with (2a)–(2d).

Proof. Assume, there exist $\mathcal{R}, s_M \mathcal{R} s_I$ and δ that satisfy conditions (2a)–(2c) of Definition 7. Define:

$$\delta'(s'_M, s'_I) := \begin{cases} \delta(s'_M, s'_I) & \text{if } \mu(s_M, s'_M) > 0 \text{ and } \varphi_u(s_I, s'_I) > 0; \\ 0 & \text{otherwise.} \end{cases}$$

Note that if $\mu(s_M, s'_M) > 0$ and $\varphi_u(s_I, s'_I) = 0$ then $\delta(s'_M, s'_I) = 0$ by (2a). Now properties (2a)–(2d) can be easily checked for δ' . □

Example 10. For Fig. 3, checking condition (2a) for t_1 boils down to $0.1 \cdot 0.3 + 0.6 \cdot 0.2 = 0.15 \in [0.1, 0.2]$. Checking condition (2b) for s_1 requires $0.7 + 0.3 = 1$.

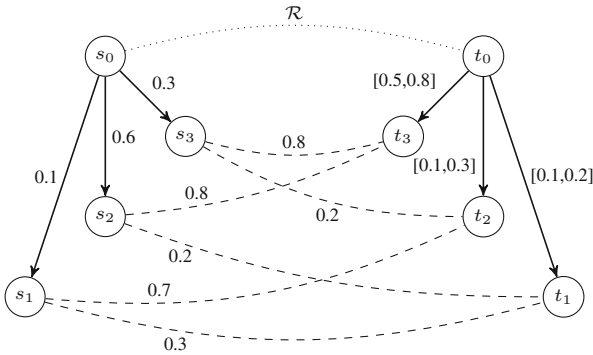


Fig. 3. The Markov Chain (left) implements the Interval Markov Chain (right)

3 Consistency of Interval Markov Chains

In this section we study consistency of Interval Markov Chains; we will return to Parametric IMCs in Sect. 5. Intuitively, an IMC is consistent if it can be implemented by at least one MC. From now on, we will drop the labelling V , since it plays no role in the discussion on consistency, and thus consider an arbitrary IMC $\mathcal{I} = (S, s_0, \varphi)$.

3.1 Local Consistency

Local consistency of a state $s \in S$ is defined with respect to a set $X \subseteq S$ of its successors, and its probability constraint $\varphi(s, \cdot)$. It ensures the existence of some probability distribution satisfying the interval constraints on transitions from s to X : the collective upper bound should be greater than or equal to 1 (condition $up(\varphi, s, X)$), the collective lower bound less than or equal to 1 ($low(\varphi, s, X)$). Moreover, each lower bound should be smaller than the corresponding upper bound, and the states outside X should be avoidable, in the sense that they admit probability 0 ($local(\varphi, s, X)$).

Definition 11 (Local consistency). *The local consistency constraints for a state $s \in S$ and a set $X \subseteq S$ is $LC(\varphi, s, X)$ s.t.:*

$$\begin{aligned} LC(\varphi, s, X) &:= up(\varphi, s, X) \wedge low(\varphi, s, X) \wedge local(\varphi, s, X), \text{ where} \\ up(\varphi, s, X) &:= \sum_{s' \in X} \varphi_u(s, s') \geq 1 \\ low(\varphi, s, X) &:= \sum_{s' \in X} \varphi_\ell(s, s') \leq 1 \\ local(\varphi, s, X) &:= (\forall s' \in X : \varphi_\ell(s, s') \leq \varphi_u(s, s')) \wedge (\forall s' \notin X : \varphi_\ell(s, s') = 0) \end{aligned}$$

We obtain the following facts, which can be directly checked from the definitions. Note that from Lemma 12(1) and (2) it follows that we may always restrict attention to the support of (φ, s) : $LC(\varphi, s, X) \equiv LC(\varphi, s, X \cap sup(\varphi, s))$.

Lemma 12. *For $X, Y \subseteq S$:*

1. *If $X \subseteq Y$ and $LC(\varphi, s, X)$ then $LC(\varphi, s, Y)$.*
2. *If $LC(\varphi, s, X)$ then also $LC(\varphi, s, X \cap sup(\varphi, s))$.*

Proof. 1. Assume $X \subseteq Y$ and $LC(\varphi, s, X)$, hence $up(\varphi, s, X)$, $low(\varphi, s, X)$ and $local(\varphi, s, X)$.

From $up(\varphi, s, X)$, we have $\sum_{s' \in X} \varphi_u(s, s') \geq 1$, so we get $up(\varphi, s, Y)$:

$$\sum_{s' \in Y} \varphi_u(s, s') = \left(\sum_{s' \in X} \varphi_u(s, s') + \sum_{s' \in Y \setminus X} \varphi_u(s, s') \right) \geq \left(1 + \sum_{s' \in Y \setminus X} \varphi_u(s, s') \right) \geq 1$$

From $local(\varphi, s, X)$, we have $\forall s' \in Y \setminus X : \varphi_\ell(s, s') = 0$, and from $low(\varphi, s, X)$, we have $\sum_{s' \in X} \varphi_\ell(s, s') \leq 1$, so we get $low(\varphi, s, Y)$:

$$\sum_{s' \in Y} \varphi_\ell(s, s') = \left(\sum_{s' \in X} \varphi_\ell(s, s') + \sum_{s' \in Y \setminus X} \varphi_\ell(s, s') \right) = \sum_{s' \in X} \varphi_\ell(s, s') + 0 \leq 1$$

Finally, from $local(\varphi, s, X)$, it holds that for $s' \in Y$, if $s' \in X$ then $\varphi_\ell(s, s') \leq \varphi_u(s, s')$, else $\varphi_\ell(s, s') = 0$, which also implies $\varphi_\ell(s, s') \leq \varphi_u(s, s')$. If $s' \notin Y$ then $s' \notin X$, so $\varphi_\ell(s, s') = 0$. So we get $local(\varphi, s, Y)$. This proves that $LC(\varphi, s, Y)$.

2. Assume $LC(\varphi, s, X)$, hence $up(\varphi, s, X)$, $low(\varphi, s, X)$ and $local(\varphi, s, X)$. Note that if $s' \in X \setminus sup(\varphi, s)$, by definition of sup , we obtain $\varphi_u(s, s') = 0$ and by $local(\varphi, s, X)$, we obtain $\varphi_\ell(s, s') = 0$.

$$\begin{aligned} \sum_{s' \in X \cap sup(\varphi, s)} \varphi_u(s, s') &= \sum_{s' \in X} \varphi_u(s, s') - \sum_{s' \in X \setminus sup(\varphi, s)} \varphi_u(s, s') \\ &= \sum_{s' \in X} \varphi_u(s, s') - 0 \geq 1 \end{aligned}$$

$$\begin{aligned} \sum_{s' \in X \cap sup(\varphi, s)} \varphi_\ell(s, s') &= \sum_{s' \in X} \varphi_\ell(s, s') - \sum_{s' \in X \setminus sup(\varphi, s)} \varphi_\ell(s, s') \\ &= \sum_{s' \in X} \varphi_\ell(s, s') - 0 \leq 1 \end{aligned}$$

Finally, if $s' \in X \cap sup(\varphi, s)$ then $s' \in X$, so $\varphi_\ell(s, s') \leq \varphi_u(s, s')$.

Otherwise, $s' \notin X$ or $s' \in X \setminus sup(\varphi, s)$, but in both cases $\varphi_\ell(s, s') = 0$. \square

3.2 Co-inductive Definition of Global Consistency

Global consistency of (P)IMCs can be defined in several ways, e.g. co-inductively and inductively. Here, we introduce a new co-inductive definition of global consistency, as a greatest fixed point (*gfp*). We first introduce an abbreviation for the set of locally consistent states w.r.t. a set X :

Notation 13. $LC_\varphi(X) := \{s \mid LC(\varphi, s, X)\}$

Next, we define $Cons$ as the greatest fixed point of LC_φ . Intuitively, from consistent states one can keep taking locally consistent steps to other consistent states.

Definition 14 (Global consistency, co-inductive). $Cons := \text{gfp}(LC_\varphi)$.

Lemma 15. *From the definition of greatest fixed point, $Cons$ is the largest set C s.t. $C \subseteq LC_\varphi(C)$:*

1. $s \in Cons \equiv s \in LC_\varphi(Cons) \equiv s \in LC_\varphi(Cons \cap sup(\varphi, s))$
2. If $C \subseteq LC_\varphi(C)$ then $C \subseteq Cons$.

Proof. (1) holds because $Cons$ is a fixed point; the second equation uses Lemma 12(2); (2) holds because $Cons$ is the *greatest* fixed point. (Tarski) \square

We motivate the definition of consistency by the following two theorems:

Theorem 16. *Let $\mathcal{M} = (S_{\mathcal{M}}, s_{\mathcal{M}0}, \mu)$ be a MC, $\mathcal{I} = (S_{\mathcal{I}}, s_{\mathcal{I}0}, \varphi)$ an IMC, and assume $\mathcal{M} \models \mathcal{I}$. Then $s_{\mathcal{I}0} \in Cons$.*

Proof. Since $\mathcal{M} \models \mathcal{I}$, there is a simulation relation \mathcal{R} , with $s_{\mathcal{M}0} \mathcal{R} s_{\mathcal{I}0}$, and for all $s_{\mathcal{M}} \in S_{\mathcal{M}}, s_{\mathcal{I}} \in S_{\mathcal{I}}$, if $s_{\mathcal{M}} \mathcal{R} s_{\mathcal{I}}$, there is a correspondence δ , satisfying properties (2a)–(2d) of Definition 7. We will prove that $\{s_{\mathcal{I}} \mid \exists s_{\mathcal{M}} : s_{\mathcal{M}} \mathcal{R} s_{\mathcal{I}}\} \subseteq \text{Cons}$. Since $s_{\mathcal{M}0} \mathcal{R} s_{\mathcal{I}0}$, it will follow that $s_{\mathcal{I}0} \in \text{Cons}$.

We proceed using the *gfp*-property (Lemma 15(2)), so it is sufficient to prove:

$$\{s_{\mathcal{I}} \in S_{\mathcal{I}} \mid \exists s_{\mathcal{M}} \in S_{\mathcal{M}} : s_{\mathcal{M}} \mathcal{R} s_{\mathcal{I}}\} \subseteq LC_{\varphi}(\{s_{\mathcal{I}} \in S_{\mathcal{I}} \mid \exists s_{\mathcal{M}} \in S_{\mathcal{M}} : s_{\mathcal{M}} \mathcal{R} s_{\mathcal{I}}\}).$$

Let $s_{\mathcal{I}} \in S_{\mathcal{I}}$ be given with $s_{\mathcal{M}} \in S_{\mathcal{M}}$ s.t. $s_{\mathcal{M}} \mathcal{R} s_{\mathcal{I}}$. Define $X := \{s'_{\mathcal{I}} \in S_{\mathcal{I}} \mid \exists s'_{\mathcal{M}} \in S_{\mathcal{M}} : \delta(s'_{\mathcal{M}}, s'_{\mathcal{I}}) > 0\}$. Clearly, if $s'_{\mathcal{I}} \in X$, then for some $s'_{\mathcal{M}} \in S_{\mathcal{M}}, \delta(s'_{\mathcal{M}}, s'_{\mathcal{I}}) > 0$ and by the correspondence property of Definition 7(2c), $s'_{\mathcal{M}} \mathcal{R} s'_{\mathcal{I}}$. So $X \subseteq \{s_{\mathcal{I}} \in S_{\mathcal{I}} \mid \exists s_{\mathcal{M}} \in S_{\mathcal{M}} : s_{\mathcal{M}} \mathcal{R} s_{\mathcal{I}}\}$. Thus, by monotonicity, Lemma 12(1), it is sufficient to show that $s_{\mathcal{I}} \in LC_{\varphi}(X)$.

To check that $s_{\mathcal{I}} \in LC_{\varphi}(X)$, we first check that the corresponding transitions yield a probability distribution:

$$\begin{aligned} & \sum_{s'_{\mathcal{I}} \in X} \sum_{s'_{\mathcal{M}} \in S_{\mathcal{M}}} \mu(s_{\mathcal{M}}, s'_{\mathcal{M}}) \cdot \delta(s'_{\mathcal{M}}, s'_{\mathcal{I}}) \\ &= \sum_{s'_{\mathcal{I}} \in S_{\mathcal{I}}} \sum_{s'_{\mathcal{M}} \in S_{\mathcal{M}}} \mu(s_{\mathcal{M}}, s'_{\mathcal{M}}) \cdot \delta(s'_{\mathcal{M}}, s'_{\mathcal{I}}) \quad (\text{if } s'_{\mathcal{I}} \notin X, \delta(s'_{\mathcal{M}}, s'_{\mathcal{I}}) = 0 \text{ by def. of } X) \\ &= \sum_{s'_{\mathcal{M}} \in S_{\mathcal{M}}} \mu(s_{\mathcal{M}}, s'_{\mathcal{M}}) \cdot (\sum_{s'_{\mathcal{I}} \in S_{\mathcal{I}}} \delta(s'_{\mathcal{M}}, s'_{\mathcal{I}})) \quad (\text{by } \sum\text{-manipulation}) \\ &= \sum_{s'_{\mathcal{M}} \in S_{\mathcal{M}}} \mu(s_{\mathcal{M}}, s'_{\mathcal{M}}) \cdot 1 \quad (\delta \text{ is a prob. distrib. by Definition 7(2b)}) \\ &= 1 \quad (\mu \text{ is a prob. dist. by Definition 1 of MC}) \end{aligned}$$

By Definition 7(2a), $\forall s'_{\mathcal{I}} \in X, \varphi_{\ell}(s_{\mathcal{I}}, s'_{\mathcal{I}}) \leq \sum_{s'_{\mathcal{M}} \in S_{\mathcal{M}}} \mu(s_{\mathcal{M}}, s'_{\mathcal{M}}) \cdot \delta(s'_{\mathcal{M}}, s'_{\mathcal{I}}) \leq \varphi_u(s_{\mathcal{I}}, s'_{\mathcal{I}})$. By the computation above, $\sum_{s'_{\mathcal{I}} \in X} \varphi_{\ell}(s_{\mathcal{I}}, s'_{\mathcal{I}}) \leq 1$ and $\sum_{s'_{\mathcal{I}} \in X} \varphi_u(s_{\mathcal{I}}, s'_{\mathcal{I}}) \geq 1$, proving *low*($\varphi, s_{\mathcal{I}}, X$) and *up*($\varphi, s_{\mathcal{I}}, X$), respectively. For $s'_{\mathcal{I}} \in X$ we already established $\varphi_{\ell}(s_{\mathcal{I}}, s'_{\mathcal{I}}) \leq \varphi_u(s_{\mathcal{I}}, s'_{\mathcal{I}})$. For $s'_{\mathcal{I}} \notin X$, by definition of X : $\forall s'_{\mathcal{M}} : \delta(s'_{\mathcal{M}}, s'_{\mathcal{I}}) = 0$. Thus, $\varphi_{\ell}(s_{\mathcal{I}}, s'_{\mathcal{I}}) \leq 0$ by the computation above, proving *local*($\varphi, s_{\mathcal{I}}, X$). This proves $s_{\mathcal{I}} \in LC_{\varphi}(X)$. \square

Conversely, we prove that a consistent IMC can be implemented by at least one MC. Note that the proof of Theorem 17 provides the construction of a concrete MC.

Theorem 17. *For IMC $\mathcal{I} = (S, s_0, \varphi)$, if $s_0 \in \text{Cons}$, then there exist a probability distribution μ and a MC $\mathcal{M} = (\text{Cons}, s_0, \mu)$ such that $\mathcal{M} \models \mathcal{I}$.*

Proof. Assume \mathcal{I} is consistent. Consider an arbitrary state $s \in \text{Cons}$. We will define $\mu(s, s')$ in between $\varphi_{\ell}(s, s')$ and $\varphi_u(s, s')$, scaling it by a factor p such that $\mu(s)$ sums up to 1. Define $L := \sum_{s' \in \text{Cons}} \varphi_{\ell}(s, s')$ and $U := \sum_{s' \in \text{Cons}} \varphi_u(s, s')$. Set $p := \frac{1-L}{U-L}$ (or 0 if $L = U$). Finally, we define $\mu(s, s') := (1-p) \cdot \varphi_{\ell}(s, s') + p \cdot \varphi_u(s, s')$.

By Lemma 15, $s \in LC_{\varphi}(\text{Cons})$, thus $L \leq 1 \leq U$. Hence $0 \leq p \leq 1$, and indeed $\varphi_{\ell}(s, s') \leq \mu(s, s') \leq \varphi_u(s, s')$. We check that $\mu(s)$ is a probability distribution:

$$\begin{aligned} \sum_{s' \in \text{Cons}} \mu(s, s') &= \sum_{s' \in \text{Cons}} ((1-p) \cdot \varphi_{\ell}(s, s') + p \cdot \varphi_u(s, s')) \\ &= (1-p)L + pU = L + p(U-L) = L + \frac{1-L}{U-L}(U-L) = 1 \end{aligned}$$

Finally, we show that \mathcal{M} implements \mathcal{I} . Define $\mathcal{R} := \{(s, s) \mid s \in \text{Cons}\}$. Define $\delta(s', s') := 1$ and $\delta(s', s'') := 0$ for $s' \neq s''$. Properties (2a)–(2c) of Definition 7 follow directly from the definition of δ , so by Lemma 9, $\mathcal{M} \models \mathcal{I}$. \square

3.3 Inductive n -Consistency

Next, we define n -consistency for a state $s \in S$ in an IMC, inductively. This rephrases the definition from [12]. Intuitively, n -consistent states can perform n consistent steps in a row. That is, they can evolve to $(n - 1)$ -consistent states.

Definition 18 (Consistency, inductive). Define sets $Cons_n \subseteq S$ by recursion on n :

$$\begin{aligned} Cons_0 &= S, \\ Cons_{n+1} &= LC_\varphi(Cons_n). \end{aligned}$$

Lemma 19. We have the following basic facts on local consistency:

1. $Cons_{n+1} \subseteq Cons_n$
2. If $m \leq n$ then $Cons_n \subseteq Cons_m$.
3. If $Cons_{n+1} = Cons_n$ and $s \in Cons_n$ then $\forall m : s \in Cons_m$.

Proof

1. Induction on n . $n = 0$ is trivial. Let $s' \in Cons_{n+2} = LC_\varphi(Cons_{n+1})$. By induction hypothesis, $Cons_{n+1} \subseteq Cons_n$, so by the monotonicity Lemma 12(1), $s' \in LC_\varphi(Cons_n) = Cons_{n+1}$, indeed.
2. Induction on $n - m$, using (1).
3. If $m > n$, we prove the result with induction on $m - n$. Otherwise the result follows from (2). □

Next, we show that universal n -consistency coincides with global consistency. Note that this depends on the assumption that the system is finitely branching.

Theorem 20. Let $sup(\varphi, s)$ be finite for all s . Then $s \in Cons \equiv \forall n : s \in Cons_n$.

Proof. \Rightarrow : Induction on n . The base case is trivial. Assume $s \in Cons$. Then $s \in LC_\varphi(Cons)$. By induction hypothesis, $Cons \subseteq Cons_n$. So, by monotonicity, Lemma 12(1), $s \in LC(Cons_n) = Cons_{n+1}$.

\Leftarrow : Assume that $s \in \bigcap_{n \geq 0} Cons_n$. Define $Y_n := Cons_n \cap sup(\varphi, s)$. By Lemma 19(1), $s \in Cons_{n+1} = LC(Cons_n) = LC_\varphi(Y_n)$ and Y_n is a decreasing sequence of finite sets (since φ has finite support). Hence it contains a smallest member, say Y_m . For Y_m , we have $s \in LC_\varphi(Y_m)$ and $Y_m \subseteq \bigcap_{n \geq 0} Cons_n$. By monotonicity, Lemma 12(1), $s \in LC_\varphi(\bigcap_{n \geq 0} Cons_n)$. So we found another fixed point, and by Lemma 15(2), $\bigcap_{n \geq 0} Cons_n \subseteq Cons$, so indeed $s \in Cons$. □

The following example shows that the condition on finite branching is essential. The situation is similar to the equivalence of the projective limit model with the bisimulation model in process algebras [3].

Example 21. Let $t \xrightarrow{[0,1]} t_i, \forall i$ and $t_{i+1} \xrightarrow{[0,1]} t_i$, see Fig. 4. Then $\forall i : t_i$ is i -consistent, but not $(i + 1)$ -consistent (since no transition exits t_0). So t is n -consistent for all n . However, no t_i is globally consistent, and $LC(t, \emptyset) = \perp$, so t is not globally consistent either.

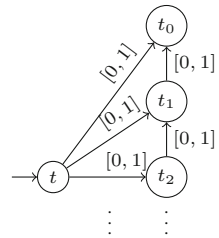


Fig. 4. Infinitely-branching IMC

Finally, we would like to limit the n that we need to check. It has been shown before [12] that when the number of states is finite, $n = |S|$ is sufficient. We now show that we can further bound the computation to the length of the longest simple path.

Definition 22 (Paths properties). *We define a (reverse) path of length n as a sequence s_0, \dots, s_{n-1} , such that for all $0 \leq i < n$, $\varphi_u(s_{i+1}, s_i) > 0$. The path is simple if for all i, j , with $0 \leq i < j < n$, we have $s_i \neq s_j$. This path is bounded by m if $n \leq m$.*

Note that, for instance, a path (s_0, s_1, s_2) has length 3 according to this definition. The essential argument is provided by the following lemma:

Lemma 23. *If $s \in \text{Cons}_{n+1} \setminus \text{Cons}_{n+2}$, then there exists a s' , with $\varphi_u(s, s') > 0$, such that $s' \in \text{Cons}_n \setminus \text{Cons}_{n+1}$.*

Proof. Assume $s \in \text{Cons}_{n+1}$ and $s \notin \text{Cons}_{n+2}$. By Lemma 12(2): $s \in LC_\varphi(\text{Cons}_n \cap \text{sup}(\varphi, s))$. Now if $\text{Cons}_n \cap \text{sup}(\varphi, s) \subseteq \text{Cons}_{n+1}$, by monotonicity we would have $s \in LC_\varphi(\text{Cons}_n \cap \text{sup}(\varphi, s)) \subseteq LC_\varphi(\text{Cons}_{n+1}) = \text{Cons}_{n+2}$, which contradicts the assumption. So there must be some $s' \in \text{Cons}_n$ with $\varphi_u(s, s') > 0$ and $s' \notin \text{Cons}_{n+1}$. \square

Since $\text{Cons}_{n+1}(s)$ depends on Cons_n of its direct successors only, consistency information propagates along paths. In particular, it propagates no longer than the longest simple path. This can be made more formal as follows:

Theorem 24. *If all simple paths from s are bounded by m , then:*

$$\text{Cons}_m(s) \equiv \forall n : \text{Cons}_n(s)$$

Proof. We first prove the following statement by induction on n :

(*) If $s \in \text{Cons}_n \setminus \text{Cons}_{n+1}$, then there exists a simple path from s of length $n + 1$, $s_0, \dots, s_n = s$, such that for all $0 \leq i \leq n$, $s_i \notin \text{Cons}_{i+1}$.

Case $n = 0$: we take the path $[s]$. Case $n + 1$: Let $s \in \text{Cons}_{n+1}$ but $s \notin \text{Cons}_{n+2}$. By Lemma 23, we obtain some s' with $\varphi_u(s, s') > 0$ and $s' \in \text{Cons}_n \setminus \text{Cons}_{n+1}$. By induction hypothesis, we get a simple path $s_0, \dots, s_n = s'$, with for all $0 \leq i \leq n$, $s_i \notin \text{Cons}_{i+1}$. Extend this path with $s_{n+1} := s$. We must show that the extended path is still simple, i.e. $s_i \neq s$ for $i \leq n$. Since $s \in \text{Cons}_{n+1}$, by Lemma 19(2), $s \in \text{Cons}_{i+1}$ but $s_i \notin \text{Cons}_{i+1}$, so $s \neq s_i$.

Finally, to prove the theorem, assume $s \in \text{Cons}_m$, with all simple paths from s bounded by m . We prove $s \in \text{Cons}_n$ by induction on n . If $n \leq m$, $s \in \text{Cons}_n$ by Lemma 19(2). Otherwise, assume as induction hypothesis $s \in \text{Cons}_n$. Note that there is no simple path from s of length $n + 1$ since $n + 1 > m$. By the statement (*), we obtain $s \in \text{Cons}_{n+1}$. So $\forall n : s \in \text{Cons}_n$. \square

To summarize, if the longest simple path from s_0 in $\mathcal{I} = (S, s_0, \varphi)$ is of length m , we can compute $s_0 \in \text{Cons}_m$. From Theorem 24, it follows that $\forall n : s_0 \in \text{Cons}_n$. By Theorem 20, we then obtain $s_0 \in \text{Cons}$. By Theorem 17 we then know that there exists a Markov Chain $\mathcal{M} = (S, s_0, \mu)$, s.t. $\mathcal{M} \models \mathcal{I}$. Conversely, if there exists any $\mathcal{M}' = (S', s'_0, \mu')$ s.t. $\mathcal{M}' \models \mathcal{I}$, we know by Theorem 16 that \mathcal{I} is consistent.

Example 25. Let us consider again the IMC in Fig. 2b. The longest simple paths from state 0 are $[0, 2, 1, 3]$ and $[0, 2, 4, 3]$, of length 4. Hence to check consistency of the IMC, it suffices to check that state 0 is in Cons_4 .

4 Algorithms for Consistency Checking of IMCs

In this section, the developed theory is used to provide algorithms to check the consistency of a finite IMC (S, s_0, φ) (i.e. without parameters). In the next section, we will synthesize parameters for PIMCs that guarantee their consistency. For IMCs, we present a forward algorithm and a backward algorithm, which are both polynomial. The justification of these algorithms is provided by the following Corollary to Lemma 12 and Definitions 14 and 18.

Corollary 26. *Let IMC (S, s_0, φ) be given, let $s \in S$ and $n \in \mathbb{N}$.*

1. $s \in \text{Cons}_{n+1} \equiv s \in LC_\varphi(\text{Cons}_n \cap \text{sup}(\varphi, s))$.
2. $s \in \text{Cons} \equiv s \in LC_\varphi(\text{Cons} \cap \text{sup}(\varphi, s))$.

The backward variant (Algorithm 1) follows our simple co-inductive definition, rephrased as Corollary 26(2). Initially, it is assumed that all states are consistent (**true**), which will be actually checked by putting them in the work list Q . When some state s is not locally consistent (LC , according to Definition 11), it is marked **false** (line 5) and all predecessors t of s that are still considered consistent, are put back in the work list Q (lines 7–9).

The forward Algorithm 2 is based on the inductive definition of consistency, rephrased in Corollary 26(1). It is called with a consistency level m and a state s to check if $s \in \text{Cons}_m$. It stores and reuses previously computed results, to avoid unnecessary computations. In particular, for each state, we store both the *maximal* level of consistency demonstrated so far (in field pc , positive consistency, initially 0), and the *minimal* level of inconsistency (in field nc , negative consistency, initially ∞). We exploit monotonicity to reuse these cached results: By Lemma 19(2), if a state is n -consistent for some $n \geq m$, then it is m -consistent as well (lines 1–2 of Algorithm 2). By contraposition, if a state is not n -consistent for $n \leq m$, then it cannot be m -consistent (lines 3–4). In all other cases, all successors $t \in \text{sup}(\varphi, s)$ are recursively checked and the $(m - 1)$ -consistent ones are collected in C (lines 5–8). Finally, we check the local consistency (LC) of C (line 9), and store and return the result of the computation in lines 9–14.

Lemma 27. *Let $|S|$ be the number of states of the IMC and let $|\varphi| := |\{(s, t) \mid \varphi_u(s, t) > 0\}|$ be the number of edges. Let d be the maximal degree $d := \max_s |\text{sup}(\varphi, s)|$.*

Algorithm 1. Consistency (backward)
 $consistent(s)$

Require: $t.cons = \mathbf{true} \ (\forall t \in S)$

```

1:  $Q := S$ 
2: while  $Q \neq \emptyset$  do
3:   pick  $r$  from  $Q$ ;  $Q := Q \setminus \{r\}$ 
4:    $X := \{t \in sup(\varphi, r) \mid t.cons = \mathbf{true}\}$ 
5:   if  $\neg LC(\varphi, r, X)$  then
6:      $r.cons := \mathbf{false}$ 
7:     for all  $t$  s.t.  $r \in sup(\varphi, t)$  do
8:       if  $t.cons = \mathbf{true}$  then
9:          $Q := Q \cup \{t\}$ 
10: return  $s.cons$ 
    
```

Co-inductive (backward) and inductive (forward) algorithms for checking consistency of IMCs.

Algorithm 2. Consistency (forward)
 $consistent(m)(s)$

Require: $t.pc = 0, t.nc = \infty \ (\forall t \in S)$

```

1: if  $m \leq s.pc$  then
2:   return true
3: else if  $m \geq s.nc$  then
4:   return false
5:  $C := \emptyset$ 
6: for all  $t \in sup(\varphi, s)$  do
7:   if  $consistent(m-1)(t)$  then
8:      $C := C \cup \{t\}$ 
9: if  $LC(\varphi, s, C)$  then
10:   $s.pc := m$ 
11:  return true
12: else
13:   $s.nc := m$ 
14:  return false
    
```

1. Algorithm 1 has worst-case time complexity $\mathcal{O}(|\varphi| \cdot d)$ and space complexity $\mathcal{O}(|S|)$.
2. Algorithm 2 has worst-case time complexity $\mathcal{O}(|S| \cdot |\varphi|)$ and space complexity $\mathcal{O}(|S| \cdot \log_2 |S|)$.

Proof. Note that in Algorithm 1 every state is set to \perp at most once. So every state s is added to Q at most $|\varphi_u(s)| + 1 = \mathcal{O}(d)$ times. Handling a state requires checking its incoming and outgoing edges, leading to $\mathcal{O}(|\varphi| \cdot d)$ time complexity. Only one bit per state is stored. Algorithm 2 is called at most $m \leq |S|$ times per state. Every call inspects the outgoing edges a constant number of times, leading to time complexity $\mathcal{O}(|S| \cdot |\varphi|)$. It stores two integers, which are at most $|S|$, which requires $2\lceil \log_2 |S| \rceil + 1$ bits. \square

Algorithm 2 could start at m equal to the length of the longest simple path, which cannot be efficiently computed in general. Algorithm 1 does not require computing any bound.

5 Parameter Synthesis for Parametric IMCs

In this section, we reconsider intervals with parameters from P . In particular, we present algorithms to synthesize the exact constraints on the parameters for which a PIMC is consistent. Note that given a PIMC and concrete values for all its parameters, we actually obtain the corresponding IMC, by just replacing the parameters by their values. This allows us to reuse all theoretical results on consistency from Sect. 3 on IMCs.

In particular, we can view parameters as “logical variables”, and view a PIMC as a collection of IMCs. For instance, consider a PIMC (P, S, s_0, φ) with

parameters $P = \{p, q\}$. Given a state s and a finite set of states $X \subseteq S$, the expression $LC(\varphi, s, X)$ can be viewed as a predicate over the logical variables p and q (appearing as parameters in φ). Also $Cons(s)$ and $Cons_n(s)$ can be viewed as predicates over p and q . In PVS, we can use universal quantification to lift results from IMCs to PIMCs. For instance, for PIMCs over P , Lemma 19.1 reads: $\forall p, q : \forall s \in S : Cons_{n+1}(s) \Rightarrow Cons_n(s)$.

Inspecting the expression $LC(\varphi, s, X)$ in Definition 11, one realizes that it is actually a constraint over p and q in linear arithmetic. However, due to the recursive/inductive nature of Definitions 14 and 18, $Cons$ and $Cons_n$ are not immediately in the form of a Boolean combination over linear arithmetic constraints. We can rephrase the definition using an enumeration over all subsets of successors, similar to [12]. Since we consider finite PIMCs, the enumeration $\exists X \subseteq S$ corresponds to a finite disjunction. Doing this we get the following variation of the inductive and co-inductive consistency definition:

Corollary 28. *Let IMC (S, s_0, φ) be given, and let $s \in S$, $n \in \mathbb{N}$*

1. $s \in Cons_{n+1} \equiv \exists X \subseteq \text{sup}(\varphi, s) : s \in LC_\varphi(X) \wedge X \subseteq Cons_n$
2. $s \in Cons \equiv \exists X \subseteq \text{sup}(\varphi, s) : s \in LC_\varphi(X) \wedge X \subseteq Cons$

Proof. We only prove (1), since (2) is similar.

\Rightarrow : Choosing $X := Cons_n$ is sufficient by Lemma 19(1).

\Leftarrow : If for some X , $s \in LC_\varphi(X \cap \text{sup}(\varphi, s))$ and $X \subseteq Cons_n$, then by monotonicity, Lemma 12(1), $s \in LC_\varphi(Cons_n \cap \text{sup}(\varphi, s))$, so $s \in Cons_{n+1}$ by Lemma 19(1). \square

It becomes clear that the synthesised parameter constraints will be Boolean combinations over linear arithmetic constraints. In particular, expressions in DNF (disjunctive normal form) provide clear insight in all possible parameter combinations. The number of combinations can be quite large, but we noticed that in many examples, most of them are subsumed by only a few maximal solutions. In particular, we will use the following notations for operators on DNFs:

- \top, \perp denote true and false (universe and empty set)
- \sqcup for their union (including subsumption simplification)
- \sqcap for their intersection (including transformation to DNF and simplification)
- \sqsubseteq for their (semantic) subsumption relation

We have experimented with two prototype realizations of our algorithms. One approach is based on CLP (constraint logic programming) in SWI-Prolog [19] + CLP(Q). We wrote a small meta-program for the subsumption check. The other approach is based on Parma Polyhedra Library [1]. In particular, its Pointset Powerset Closed Polyhedra provide efficient implementations of the operations on DNFs.

5.1 Inductive Approach

In the inductive approach, Corollary 28(1) gives rise to a set of equations on variables $v_{n,s}$ (state s is n -consistent):

$$\begin{aligned} \text{let } v_{0,s} &= \top \\ \text{let } v_{n+1,s} &= \bigsqcup_{X \subseteq \text{sup}(\varphi,s)} (LC_\varphi(X) \sqcap \prod_{t \in X} v_{n,t}) \end{aligned}$$

Starting from the initial state $v_{|S|,s_0}$, we only need to generate the reachable equations that are not pruned away by inconsistent $LC_\varphi(X)$ constraints, and we need to compute each equation only once. Note that there will be at most $|S|^2$ reachable equations. The number of conjunctions per equation is bounded by 2^d , the number of $X \subseteq \text{sup}(\varphi, s)$, and for each X we build a conjunction of length $O(d)$. So, the size of the whole set of equations is bounded by $O(|S|^2 \cdot d \cdot 2^d)$. In general, however, there is no polynomial upper bound on the size of the corresponding solution. Also, note that by Theorem 24, we could replace $|S|$ by the length of the longest simple path.

The set of equations can be interpreted directly as a Constraint Logic Program in Prolog, with predicates $\text{cons}(N, S)$. Prolog will compute the dependency tree for $s \in \text{Cons}_n$, by backtracking over all choices for X . Along the way, all encountered LC_φ predicates are asserted as constraints to the CLP solver. This has the advantage that locally inconsistent branches will be pruned directly, without ever generating their successors. By enumerating all feasible solutions, we obtain the complete parameter space as a disjunction of conjunctive constraints.

However, the computation tree has a lot of duplicates, and the number of returned results is very high, since we start out with a deeply nested and-or-expression. We provide a more efficient version in Algorithm 3. Here recomputations are avoided by caching all intermediate results in a *Table* (see lines 3 and 12). For each enumerated subset of successors X (lines 12), the algorithm checks $(n-1)$ -consistency. Note that we “shortcut” this computation as soon as we find that either X is not locally consistent, or some $t \in X$ is not $(n-1)$ -consistent (line 8). The final optimization is to suppress all subsumed results in the resulting conjunctions and disjunctions. We show this by using \sqcap and \sqcup . This drastically reduces the number of returned disjunctions.

We have implemented Algorithm 3 in Prolog+CLP, using meta-programming techniques to suppress subsumed results. Alternatively, one could implement the algorithm directly on top of the Parma Polyhedra Library.

5.2 Co-inductive Approach

Next, we show how to encode co-inductive consistency in a Boolean Equation System (BES) (over sets of polyhedra). Here, the equations will be recursive. The largest solution for variable v_i indicates that state s_i is consistent. This solution provides then a description of the set of all parameters for which the PIMC is consistent.

Algorithm 3. Inductive Parameter Synthesis Algorithm $Cons(s, n)$

Require: Initialize: $Table := \emptyset$

- 1: **if** $n = 0$ **then**
- 2: **return** \top
- 3: **if** $\exists R : (s, n, R) \in Table$ **then**
- 4: **return** R
- 5: $D := \perp$
- 6: **for all** $X \subseteq sup(\varphi, s)$ **do**
- 7: $C := LC_\varphi(X)$
- 8: **while** $X \neq \emptyset \wedge C \neq \perp$ **do**
- 9: pick t from X ; $X := X \setminus \{t\}$
- 10: $C := C \sqcap Cons(t, n - 1)$
- 11: $D := D \sqcup C$
- 12: add (s, n, D) to $Table$
- 13: **return** D

Algorithm 4. Co-inductive Parameter Synthesis Algorithm

Require: Initialize: $s.sol := \top (\forall s \in S)$

- 1: $Q := S$
- 2: **while** $Q \neq \emptyset$ **do**
- 3: pick s from Q ; $Q := Q \setminus \{s\}$
- 4: $sol := \bigsqcup_{X \subseteq sup(\varphi, s)} (LC_\varphi(X) \sqcap \prod_{t \in X} t.sol)$
- 5: **if** $sol \sqsubseteq s.sol$ **then**
- 6: $s.sol := sol$
- 7: **for all** t s.t. $s \in sup(\varphi, t)$ **do**
- 8: **if** $t.sol \neq \perp$ **then**
- 9: $Q := Q \cup \{t\}$
- 10: **return** $s_0.sol$

Inductive and co-inductive algorithms for parameter synthesis in PIMCs.

Definition 29 (BES in DNF). Given a PIMC $\mathcal{P} = (P, S, s_0, \varphi)$, we define the BES as the following set of equations, for each formal variable $v_s, s \in S$:

$$\left\{ v_s = \bigsqcup_{X \subseteq sup(\varphi, s)} (LC_\varphi(X) \sqcap \prod_{t \in X} v_t) \mid s \in S \right\}$$

We can bound the size of this BES and the number of iterations for its solution. Again, the size of the final solution cannot be bounded polynomially.

Lemma 30. For each PIMC $\mathcal{P} = (P, S, s_0, \varphi)$, with out-degree bounded by d , the corresponding BES has size $O(|S|.d.2^d)$. The BES can be solved in $O(\ell)$ iterations, where ℓ is the length of the longest simple path in \mathcal{P} .

Proof. We have $|S|$ equations of size at most $O(d.2^d)$ each. The BES can be solved by value iteration. Let F_s denote the right hand side of the equation for v_s . We can compute the largest solution by iteration and substitution as follows:

$$\begin{aligned} \sigma_0 &= \lambda s. \top \\ \sigma_{n+1} &= \lambda s. F_s[v_t \mapsto \sigma_n(t) \mid t \in S] \end{aligned}$$

By monotonicity, $\sigma_n \supseteq \sigma_{n+1}$ (pointwise). We can terminate whenever $\sigma_n \subseteq \sigma_{n+1}$. Since it can be proved by induction that $\sigma_n \equiv Cons_n$, the process terminates within ℓ steps by Lemma 24. \square

Solving this BES can be done by straightforward value iteration, see Lemma 30. The intermediate expressions can be viewed as collections of polyhedra, represented e.g. by the Parma Polyhedra Library as Powersets of Convex Polyhedra [1]. Algorithm 4 provides a variant of the iteration in Lemma 30. Here we only update the polyhedra for nodes whose successors have been modified. To

this end, we maintain a worklist Q of states that we must check initially (line 1) or when their successors are updated (line 9). This algorithm can be viewed as the parametric variant of the backward Algorithm 1.

Example 31. On the example in Fig. 2a, Definition 29 gives rise to the following equation system (simplifying trivial arithmetic inequalities and global bounds $0 \leq p, q \leq 1$).

$$\begin{array}{ll}
 v_0 = v_1 & v_2 = v_1 \ \& \ p=1 \\
 \quad | \ v_2 & \quad | \ v_2 \ \& \ q=1 \\
 \quad | \ v_1 \ \& \ v_2 & \quad | \ v_1 \ \& \ v_2 \ \& \ p+q=1 \\
 v_1 = v_1 \ \& \ v_3 \ \& \ q \geq 0.3 \ \& \ q < 0.7 & \quad | \ v_1 \ \& \ v_4 \ \& \ p \geq 0.5 \\
 v_3 = v_3 & \quad | \ v_2 \ \& \ v_4 \ \& \ q \geq 0.5 \\
 v_4 = \text{false} & \quad | \ v_1 \ \& \ v_2 \ \& \ v_4 \ \& \ p+q \geq 0.5
 \end{array}$$

We solve the simplified BES by value iteration as follows. In Approximation 0, each $v_i = \text{true}$. We show approximations 1–3, which is the stable solution. From the final result, we can conclude that the initial state in Fig. 2a is consistent, if and only if $0.3 \leq q \leq 0.7 \vee q = 1$.

Approximation: 1	Approximation: 2	Approximation: 3
$v_0 = \text{true}$	$v_0 = p+q \geq 0.5$	$v_0 = q \geq 0.3 \ \& \ q < 0.7$
	$ \ q \geq 0.3 \ \& \ q < 0.7$	$ \ q = 1$
$v_1 = q < 0.7 \ \& \ q \geq 0.3$	$v_1 = q \geq 0.3 \ \& \ q < 0.7$	$v_1: \text{idem}$
$v_2 = p+q \geq 0.5$	$v_2 = p+q = 1 \ \& \ q \geq 0.3 \ \& \ q < 0.7$	$v_2: \text{idem}$
	$ \ q = 1$	
$v_3 = \text{true}$	$v_3 = \text{true}$	$v_3 = \text{true}$
$v_4 = \text{false}$	$v_4 = \text{false}$	$v_4 = \text{false}$

6 Experiments

To get an indication of the effectiveness of our algorithms and optimizations, we performed some experiments in a Prolog prototype implementation on Lehmann and Rabin’s randomized dining philosophers from Prism [15]. First, we modified that DTMC to an IMC by replacing probabilities like 0.1666 by intervals $[0.1, 0.2]$. This made the consistency analysis numerically more stable. Subsequently, we replaced some probabilities by parameters to get a PIMC, and experimented with different intervals as featured in the first column of Table 1, for one parameter P or two parameters P and Q . The number of edges with these parameters is given in the third column of the table. We compared the inductive algorithm for increasing n -consistency with the co-inductive algorithm. Column CLP shows our Prolog implementation of the inductive algorithm from [12]. CLP+opt corresponds to the inductive Algorithm 3, including our optimizations, i.e. caching and subsumption. PPL shows our co-inductive Algorithm 4. We carried out the experiments on the model for 3 philosophers (956 states, 3625 edges) and 4 philosophers (9440 states, 46843 edges).

Table 1. Experimental results. All entries are time in seconds. Timeout was 1 h.

Interval(s)	Model	Param. edges	CLP		CLP+opt		PPL
			<i>n</i> -inductive		<i>n</i> -inductive		co-inductive
			<i>n</i> = 2	<i>n</i> = 3	<i>n</i> = 10	<i>n</i> = 50	<i>n</i> = ∞
[<i>P</i> , <i>P</i> + 0.1]	phils3	723	0.01	0.02	3.82	82.01	5.08
	phils4	14016	0.03	0.14	51.84	2506.61	360.02
[0, <i>P</i>]	phils3	723	0.29	Timeout	5.45	123.42	8.78
	phils4	14016	181.15	Timeout	82.18	Timeout	1605.94
[<i>P</i> , 1]	phils3	723	0.21	Timeout	5.02	102.36	8.67
	phils4	14016	100.57	Timeout	77.62	3300.53	1016.90
[<i>P</i> , <i>Q</i>]	phils3	723	0.35	Timeout	27.35	Timeout	10.33
	phils4	14016	472.79	Timeout	118.64	Timeout	1649.00
[0, <i>P</i>], [0.3, <i>Q</i>]	phils3	723 + 1416	0.30	Timeout	122.66	Timeout	18.33
	phils4	14016 + 688	318.91	Timeout	834.77	Timeout	2575.11
[<i>P</i> , 1], [0.3, <i>Q</i>]	phils3	723 + 1416	0.22	Timeout	13.53	893.27	13.52
	phils4	14016 + 688	161.69	Timeout	84.61	Timeout	1853.24

Table 1 shows the results; all times are measured with the SWI-Prolog library **statistics**. Increasing *n*-consistency highly impacts the performance. It is clear that caching and subsumption provide useful optimizations, since we can now compute consistency for much higher *n*. The co-inductive variant with PPL wins, since it is always faster than CLP+opt for *n* = 50. We also observe that the increase time from 3 to 4 philosophers is considerable. This is consistent with the complexity result (note that for phil4, $d \approx 5$ so $d \cdot 2^d \approx 150$).

Note that PPL computes consistency constraints for *all* states and for *all* *n*, whereas CLP only computes this for small *n* and the initial state. As an example, the constraint computed by PPL for 3 philosophers, 2 parameters, intervals [0, *P*] and [0.3, *Q*] is $(3P \geq 0.5 \wedge Q \geq 0.34) \vee (2P + Q \geq 1 \wedge p \geq 0.25 \wedge 3Q \geq 1)$.

7 Perspectives

Using inductive and co-inductive definitions of consistency, we provided forward and backward algorithms to synthesize an expression for all parameters for which a PIMC is consistent. The co-inductive variant, combined with a representation based on convex polyhedra, provides the most efficient algorithm. We believe that our work extends straightforwardly when the intervals are replaced by arbitrary linear constraints, since both CLP and PPL can handle linear constraints. The resulting formalism would generalize (linear) Constraint Markov Chains [11] by adding global parameters. Also, our approach should apply directly to parameter synthesis for consistent reachability [12].

We also plan to experiment with other case studies, e.g. the benchmarks of [2], with an implementation that is more elaborate than our initial prototype. This would give insight on how the algorithms scale up w.r.t. the number of

parameters and of parametric edges. Finally, [2] gives a polynomial size CLP with extra variables, but doesn't address the parameter synthesis problem. Polynomial size CLP programs without extra variables can be obtained using if-then-else, e.g. $(v_0?1 : 0) + (v_1?1 : 0) + (v_2?p : 0) \leq 1$. However, we don't know of a method to solve equation systems with conditionals without expanding them to large intermediate expressions.

Acknowledgement. The authors would like to thank the reviewers for their extensive comments, which helped them to improve the paper. They acknowledge the support of University Paris 13 and of the Van Gogh project PAMPAS, that covered their mutual research visits.

References

1. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.* **72**(1–2), 3–21 (2008)
2. Bart, A., Delahaye, B., Lime, D., Monfroy, É., Truchet, C.: Reachability in parametric interval Markov chains using constraints. In: Bertrand, N., Bortolussi, L. (eds.) QEST 2017. LNCS, vol. 10503, pp. 173–189. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66335-7_11
3. Bergstra, J.A., Klop, J.W.: The algebra of recursively defined processes and the algebra of regular processes. In: ICALP 1984, pp. 82–94 (1984)
4. Česka, M., Pilař, P., Paoletti, N., Brim, L., Kwiatkowska, M.: PRISM-PSY: precise GPU-accelerated parameter synthesis for stochastic systems. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 367–384. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_21
5. Chakraborty, S., Katoen, J.-P.: Model checking of open interval Markov chains. In: Gribaudo, M., Manini, D., Remke, A. (eds.) ASMTA 2015. LNCS, vol. 9081, pp. 30–42. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18579-8_3
6. Chen, T., Han, T., Kwiatkowska, M.Z.: On the complexity of model checking interval-valued discrete time Markov chains. *Inf. Process. Lett.* **113**(7), 210–216 (2013)
7. Daws, C.: Symbolic and parametric model checking of discrete-time Markov chains. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 280–294. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31862-0_21
8. Dehnert, C., Junges, S., Jansen, N., Corzilius, F., Volk, M., Bruinjtes, H., Katoen, J.-P., Ábrahám, E.: PROPhESY: a PRObabilistic ParamEter SYNthesis tool. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 214–231. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_13
9. Delahaye, B.: Consistency for parametric interval Markov chains. In: SynCoP 2015, pp. 17–32 (2015)
10. Delahaye, B., Katoen, J.-P., Larsen, K.G., Legay, A., Pedersen, M.L., Sher, F., Wasowski, A.: Abstract probabilistic automata. *Inf. Comput.* **232**, 66–116 (2013)
11. Delahaye, B., Larsen, K.G., Legay, A., Pedersen, M.L., Wasowski, A.: New results for constraint Markov chains. *Perform. Eval.* **69**(7–8), 379–401 (2012)
12. Delahaye, B., Lime, D., Petrucci, L.: Parameter synthesis for parametric interval Markov chains. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 372–390. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_18

13. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric Markov models. *STTT* **13**(1), 3–19 (2011)
14. Jonsson, B., Larsen, K.G.: Specification and refinement of probabilistic processes. In: *LICS 1991*, pp. 266–277 (1991)
15. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) *CAV 2011*. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
16. Lanotte, R., Maggiolo-Schettini, A., Troina, A.: Parametric probabilistic transition systems for system design and analysis. *Formal Asp. Comput.* **19**(1), 93–109 (2007)
17. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) *CADE 1992*. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55602-8_217
18. Quatmann, T., Dehnert, C., Jansen, N., Junges, S., Katoen, J.-P.: Parameter synthesis for Markov models: faster than ever. In: Artho, C., Legay, A., Peled, D. (eds.) *ATVA 2016*. LNCS, vol. 9938, pp. 50–67. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_4
19. Wielemakers, J.: SWI-prolog version 7 extensions. In: *WLPE-2014*, July 2014



Information Flow Tracking for Side-Effectful Libraries

Alexander Sjösten¹(✉), Daniel Hedin^{1,2}, and Andrei Sabelfeld¹

¹ Chalmers University of Technology, Gothenburg, Sweden
sjosten@chalmers.se

² Mälardalen University, Västerås, Sweden

Abstract. Dynamic information flow control is a promising technique for ensuring confidentiality and integrity of applications that manipulate sensitive information. While much progress has been made on increasingly powerful programming languages ranging from low-level machine languages to high-level languages for distributed systems, surprisingly little attention has been devoted to libraries and APIs. The state of the art is largely an all-or-nothing choice: either a *shallow* or *deep* library modeling approach. Seeking to break out of this restrictive choice, we formalize a general mechanism that tracks information flow for a language that includes higher-order functions, structured data types and references. A key feature of our approach is the *model heap*, a part of the memory, where security information is kept to enable the interaction between the labeled program and the unlabeled library. We provide a proof-of-concept implementation and report on experiments with a file system library. The system has been proved correct using Coq.

1 Introduction

While useful, access control is not enough: it is crucial what applications do with the data after access has been granted [25]. Information flow control tracks the propagation of data in programs, thus enforcing confidentiality and integrity policies. Due to the widespread use of highly dynamic languages, such as JavaScript, there has been a growing interest in *dynamic information flow control*. There are two basic kinds of flows to consider: *explicit* and *implicit* [5], related to the notions of *data flow* and *control flow*. Dynamic information flow is tracked at runtime by extending the data with *security labels*, which are propagated and checked against a *security policy* during execution. The detection of potential security violations cause program execution to halt.

While much progress has been made on increasingly powerful programming languages ranging from low-level machine languages to high-level languages for distributed systems, surprisingly little attention has been devoted to *libraries* and *APIs*¹. The main challenge is when the library is not written in the language

¹ For elegance of expression, when we write library in this paper we refer to both libraries and APIs.

itself, and thus not compatible with the labeled semantics of the program. There are mainly two situations where this occurs: (1) when the library is part of the standard execution environment, and (2) when the library is brought into the language using some form of *foreign function interface* (FFI). In such cases, values passing between the program and the library must be translated. The process of translating values from one programming language to another is known as *marshaling*.

Marshaling of labeled values additionally entails that security labels must be removed from the values being passed from the program to the library, and reattached on the values returned from the library to the program. We refer to those steps as *unlabeling* and *relabeling* of the values, and the description of how it should be done as a *library model*. The main difference between standard marshaling and marshaling of labeled values is the latter removes information from the values passed to the library. To be able to correctly relabel values going from the library to the program, the labels removed during the unlabeling process must be used, since the returned value contains no security information. This means that the library models are inherently stateful—the removed labels are stored in a *model state* used when relabeling.

Library models can be split into two categories: *deep* and *shallow* models [14]. Deep models track information flow inside the library, requiring precise modeling of the execution of the library, while shallow models are limited to the security labels on the boundary of the library. Often, deep models necessitate reimplementing parts of the library functionality within the model, making them difficult to create and maintain. Shallow models, on the other hand, are significantly more lightweight, but possibly too imprecise. In this work, we are interested in the boundary between deep and shallow models.

Current state of the art in dynamic information-flow tracking does not fit this classification entirely, in part due to ad-hoc handling of libraries. To the extent addition of new libraries is supported, the models used tend towards shallow models. This is true for, e.g., FlowFox [13], and experimental extensions of JSFlow [15]. On the other hand, JSFlow and FlowFox both use deep models to provide fine grained information-flow tracking for built in libraries. JSFlow, e.g., implements the full ECMA-262 version 5 standard using what is best considered a deep approach.

In recent work, Hedin et al. [17] initiate a framework for tracking information flow in libraries. The setting is a labeled *program* and an unlabeled *library* that share the same core semantics (*split semantics*) in order to limit the marshaling to security labels only. Their work targets a focused functional language with higher-order functions (which allows for both callbacks and promises to exist), and structured data in terms of lists. It does not, however, handle side effects, which means that many libraries cannot be modeled in a satisfactory way. As an example, it is unavoidable for a standard file system library to maintain state to keep track of open files, stream positions and buffers. The success of a function `read(path, success, fail)` is dependent both on the file path and the state of the library which must be reflected by security models for the library.

The combination of state and higher-order functions significantly complicates the library models and the model state over the ones used by Hedin et al. If the state is first-class (i.e., it can be sent around as values, as in languages with mutable references, records or objects) the situation is further complicated. *This is the setting we are interested in handling, as it captures the essence of many of the problems found when modeling real libraries.*

To this end we introduce a model heap, allowing library values to be tied to a mutable model state, which allows for secure modeling of the interaction between first-class state and higher-order functions.

Consider the file system example, depicted in Fig. 1. When the program calls the library function `read`, the library function is first lifted into the program using the corresponding function model defined by the library model, `LModel`. The lifting (illustrated by the dotted arrow in the figure) is done by means of wrapping and results in an unlabeled function that can be called by the program. When the wrapper is called with labeled arguments, a new call model state, `CModel`, is created and used to hold the labels of the arguments, since the underlying library function requires unlabeled values. As can be seen in the figure, the call model state is connected to the library model state and together they define the model state that the function model of `read` interacts with. Any other values, including higher-order functions and first-class state, defined in the library share the same library model state, which guarantees that they have the same view of the library state, even in the presence of mutability.

There are two main benefits of our work over ad-hoc modeling of libraries. First, it lowers the modeling effort significantly, and, second, given that the models properly describe the library, it guarantees noninterference. Both benefits stem from expressing the models in a simplified model language that controls the marshaling process, thus sidestepping the need to reimplement it repeatedly.

Considering the dimension of shallow and deep models, our work can be seen as exploring the boundary. Shallow models are expressed solely in terms of the boundary labels, while our work gains access to intermediate labels when models for lazy marshaling, higher-order functions and first-class state are triggered. In addition, it is relatively easy to extend our system to allow models to use the runtime values allowing for *dependent models* [17]. Compared to fully deep models, our work is limited to the information passing between the program and the library at the point of passing. Thus, intermediate values and labels that do not participate in cross-boundary activity is without reach. While deep models in theory have access to more information and therefore have the potential to be more precise, it is unclear if the added precision is significant in practice, in particular in the light of the added implementation cost.

Contributions. The main contributions of this paper are:

- We have created a language containing three cornerstones of library modeling: higher-order functions, first-class state, and structured values (the syntax

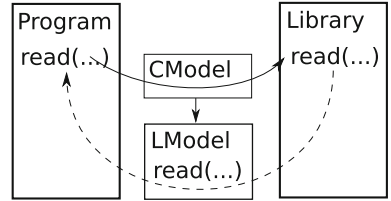


Fig. 1. Model heap illustration

and semantics are presented in Sect. 2 and Sect. 3, respectively, while Sect. 6 discusses correctness).

- We have implemented a prototype and used it to explore the interaction between the different features of the language (examples that illustrate our mechanism are reported in Sect. 4).
- We have conducted a case study on a file system library, inspired by the file system library in node.js [10], showing that our language is able to handle stateful libraries (the case study is reported in Sect. 5).
- We have formalized the language and its correctness proof in Coq [19].

The scope of the prototype is to experimentally verify applicability of models, not to assess performance in a full-scale implementation. The prototype serves as a complement to the formal proof to create a system that is both correct and useful. The full version of the paper, along with the formalization in Coq and the proof-of-concept prototype can be found at [27].

2 Syntax

The language we present is a small functional language with split semantics and lazy marshaling. The syntax of the language is defined as follows, where n denotes numbers and x denotes identifiers.

$$\begin{aligned}
 e ::= & n \mid x \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{fun } x = e \mid e_1 \ e_2 \mid \\
 & x_{lib} \mid e_1 \oplus e_2 \mid \ominus e \mid \text{head } e \mid \text{tail } e \mid e_1 : e_2 \mid [] \mid (e_1, e_2) \mid () \mid \\
 & \text{ref } e \mid !e \mid \{ \underline{x} : \underline{e} \} \mid e.x \mid e_1 := e_2 \mid e_1 ; e_2 \mid \text{upg } e \ \ell
 \end{aligned}$$

The syntax of the language is entirely standard apart from the x_{lib} construction that lifts a *library value* to a *program value*, and $\text{upg } e \ \ell$ that gives the result of the expression a given label, $\ell ::= L \mid H$. For simplicity, we identify sets with the meta variables ranging over them. Let \underline{X} range over lists of X for any set X , where $[]$ denotes the empty list and \cdot denotes the cons operator. An application in the language is a triple $(\underline{d}_p, \underline{d}_l, \underline{m})$, where the first component is the labeled *program*, the second component is the unlabeled *library* and the third component is the *library model*. Throughout the rest of this paper, we use *program* when referring to the labeled part, and *library* when referring to the unlabeled part.

The top-level definitions, d , allow for named definitions of functions and values $d ::= \text{fun } f(x) = e \mid \text{let } x = e$. The top-level model definitions, m , allow for named definitions of models and labels $m ::= \text{mod } x :: \gamma \mid \text{lbl } x :: \kappa$, where γ denotes *relabel models* and κ denotes *label terms*. The label terms, $\kappa ::= \ell \mid \alpha \mid \kappa_1 \sqcup \kappa_2$ are terms that evaluate to labels in a given model state and consist of *labels*, ℓ , *label variables*, α , and the least upper bound of two label terms. The relabel models, γ , used to relabel library values, are defined as follows

$$\gamma ::= \kappa \mid (\gamma_1, \gamma_2)^\kappa \mid [\gamma]^\kappa \mid (\varphi \rightarrow \gamma, \zeta)^\kappa \mid \text{ref}(\varphi, \gamma)^\kappa$$

where φ denotes *unlabel models*, used to unlabel program values, and ζ denotes *effect constraints* defined below. All values are given a label by a label term, and

the relabeling of structured values follows the structure of the value. To relabel a function, we must know how to unlabel the argument, how to relabel the result, and how the function interacts with the model state. To relabel a reference we must know how to unlabel the values written and how to relabel the values read. The unlabel models, φ , are defined as follows.

$$\varphi ::= \alpha \mid \#\alpha^\alpha \mid (\varphi_1, \varphi_2)^\alpha \mid [\varphi]^\alpha$$

Unlabeling of values is performed by storing the label of the value in the corresponding label variable in the model state. As for relabeling, unlabeling of structured values follows the structure of the value. Unlabeling of functions and references introduces an *abstract name*, $\#\alpha$, used by library functions to tie any interaction to their model state in the effect constraints, ζ .

$$\zeta ::= !\#\alpha \rightarrow \varphi \mid \kappa \vdash \#\alpha \leftarrow \gamma \mid \kappa \vdash \#\alpha \gamma \rightarrow \varphi \mid \kappa \vdash \alpha \leftarrow \kappa$$

In the order of definition: a library function that reads a labeled reference defines how to unlabel the read value, a library function that writes to a labeled reference defines the security context in which the write occurs and how to relabel the value to be written, a library function that calls a labeled function defines the security context in which the call occurs, how to relabel the parameter and how to unlabel the result, and finally, a library function that modifies the library state defines the security context of the update and how the security model changes.

3 Semantics

We define the semantics step-wise in three parts. The first part defines the labeled values, and the execution environment. The second part defines the evaluation relation and how the function representations of the values are created and used in the semantics. Finally, the third part defines how values are marshaled between the program and the library. For space reasons, parts of the semantic definitions have been left out. We refer the reader to the full version of this paper [27] for the missing definitions.

3.1 Values

In order to differentiate between the labeled semantics and the unlabeled semantics, we use \hat{X} to denote an entity in the labeled semantics corresponding to the entity X in the unlabeled semantics. We only give the labeled values. The unlabeled values are defined analogously. The values in the language, \hat{v} , are integers n , tuples, higher-order functions \hat{F} , lists (\hat{H}, \hat{T}) , references (\hat{R}, \hat{W}) , and records \hat{O} , where higher-order functions, lists, references and records are represented as (pairs of) functions in order to simplify the marshaling.

$$\hat{v} ::= n^\ell \mid (\hat{v}_1, \hat{v}_2)^\ell \mid ()^\ell \mid \hat{F}^\ell \mid (\hat{H}, \hat{T})^\ell \mid []^\ell \mid (\hat{R}, \hat{W})^\ell \mid \hat{O}^\ell$$

The labels, ℓ , form a two-point upper semi-lattice $L \sqsubseteq H$, where L denotes *low* (public) and H denotes *high* (private). Let $\ell_1 \sqcup \ell_2$ denote the least upper bound of ℓ_1 and ℓ_2 , and let $\hat{v}^{\ell_2} = v^{\ell_1 \sqcup \ell_2}$ for $\hat{v} = v^{\ell_1}$.

The execution environment is a triple $(\varsigma, \Gamma, \Sigma)$ of the security context, ς , the stack, and the heap. The security context ς ranges over labels ℓ . The stack Γ is a triple of stacks $(\hat{\rho}, \underline{\rho}, \ddot{\rho})$, containing pointers to the labeled frames, the unlabeled frames and the model frames, respectively. The heap Σ is a triple of heaps, $(\hat{\sigma}, \sigma, \ddot{\sigma})$, consisting of the labeled heap, the unlabeled heap and the model heap. The labeled and unlabeled heaps can contain values (for implementing references), and frames, whereas the model heap only contains frames. The labeled and unlabeled frames, $\hat{\omega}$ and ω , are maps from identifiers to values, and the model frames, $\ddot{\omega}$ are maps from identifiers to *model items*. Each frame represents a scope, and together with the corresponding stacks they form scope chains. The model items, $\ddot{i} ::= \ell \mid \gamma \mid \zeta$, consists of labels, relabel models and effect constraints.

3.2 Evaluation Relations

The evaluation relation for program execution is of the form $\varsigma, \Gamma \models (\Sigma_1, e) \rightarrow (\Sigma_2, \hat{v})$, read “expression e evaluates in the environment consisting of the security context, ς , the stack, Γ , and the heap, Σ_1 , resulting in the updated heap Σ_2 and value \hat{v} ”. Similarly, library execution is of the form $\varsigma, \Gamma \models (\Sigma_1, e) \rightsquigarrow (\Sigma_2, v)$, where the unlabeled semantics is parameterized over the security context to model that the context is global and always available to the marshaling functions².

Figure 2 contains a selection of the semantic rules of the program semantics related to the marshaling of values.

The rules of the core language are standard. Whenever an integer is created (`int`), it is always originally labeled L . Variables are retrieved from the labeled heap using `lookupL` in `var`. If-statements (`if-true` and `if-false`) evaluate the conditional expression and based on the result select which branch to take. The branch taken is evaluated in a security context of $\varsigma \sqcup \ell$ and the returned value is raised to ℓ , where ℓ is the label of the result of the conditional expression.

Function closures are represented as functions, $\hat{F} : (\varsigma, \Gamma, \Sigma_1, \hat{v}) \rightarrow (\Sigma_2, \hat{v})$, created by `lclos` (`fun`) in the following way.

$$\begin{aligned} \text{lclos}(\hat{\rho}', x, e) &= \lambda(\varsigma, (\hat{\rho}, \underline{\rho}, \ddot{\rho}), (\hat{\sigma}_1, \sigma_1, \ddot{\sigma}_1), \hat{v}_1) . (\Sigma, \hat{v}_2) \\ &\text{where } \hat{\sigma}_2 = \hat{\sigma}_1[\hat{\rho} \mapsto \{x \mapsto \hat{v}_1^{\varsigma}\}], \hat{\rho} \text{ fresh} \\ &\text{and } \varsigma, (\hat{\rho} \cdot \hat{\rho}', \underline{\rho}, \ddot{\rho}) \models ((\hat{\sigma}_2, \sigma_1, \ddot{\sigma}_1), e) \rightarrow (\Sigma, \hat{v}_2) \end{aligned}$$

The function closure will, when interacted with, create a new pointer to a labeled frame containing the mapping of the parameter name x and the actual value \hat{v}_1 , which is raised to the current security context. The function expression e is then

² In an operational semantics global non-constant values must be passed around during execution, similar to in a pure functional language.

$$\begin{array}{c}
 \text{int} \frac{}{\varsigma, \Gamma \models (\Sigma, n) \rightarrow (\Sigma, n^L)} \quad \text{var} \frac{\text{lookupL}(\Gamma, \Sigma, x) = \hat{v}}{\varsigma, \Gamma \models (\Sigma, x) \rightarrow (\Sigma, \hat{v})} \\
 \\
 \text{if-true} \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \rightarrow (\Sigma_2, v_1^\ell) \quad v_1 \neq 0 \quad \varsigma \sqcup \ell, \Gamma \models (\Sigma_2, e_2) \rightarrow (\Sigma_3, \hat{v}_2)}{\varsigma, \Gamma \models (\Sigma_1, \text{if } e_1 \text{ then } e_2 \text{ else } e_3) \rightarrow (\Sigma_3, \hat{v}_2^\ell)} \\
 \\
 \text{if-false} \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \rightarrow (\Sigma_2, v_1^\ell) \quad v_1 = 0 \quad \varsigma \sqcup \ell, \Gamma \models (\Sigma_2, e_3) \rightarrow (\Sigma_3, \hat{v}_3)}{\varsigma, \Gamma \models (\Sigma_1, \text{if } e_1 \text{ then } e_2 \text{ else } e_3) \rightarrow (\Sigma_3, \hat{v}_3^\ell)} \\
 \\
 \text{fun} \frac{}{\varsigma, (\hat{\rho}, \underline{\rho}, \ddot{\rho}) \models (\Sigma, \text{fun } x = e) \rightarrow (\Sigma, \text{lclose}(\hat{\rho}, x, e)^L)} \\
 \\
 \text{app} \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \rightarrow (\Sigma_2, \hat{F}^\ell) \quad \varsigma, \Gamma \models (\Sigma_2, e_2) \rightarrow (\Sigma_3, \hat{v}_1) \quad \hat{F}(\varsigma \sqcup \ell, \Gamma, \Sigma_3, \hat{v}_1) = (\Sigma_4, \hat{v}_2)}{\varsigma, \Gamma \models (\Sigma_1, e_1 e_2) \rightarrow (\Sigma_4, \hat{v}_2^\ell)} \\
 \\
 \text{ref} \frac{\varsigma, \Gamma \models (\Sigma, e) \rightarrow ((\hat{\sigma}, \sigma, \ddot{\sigma}), \hat{v}) \quad \hat{\rho} \text{ fresh}}{\varsigma, \Gamma \models (\Sigma, \text{ref } e) \rightarrow ((\hat{\sigma}[\hat{\rho} \mapsto \hat{v}], \sigma, \ddot{\sigma}), (\text{lread}(\hat{\rho}), \text{lwrite}(\hat{\rho}))^L)} \\
 \\
 \text{deref} \frac{\varsigma, \Gamma \models (\Sigma_1, e) \rightarrow (\Sigma_2, (\hat{R}, \hat{W})^\ell) \quad \hat{R}(\varsigma \sqcup \ell, \Gamma, \Sigma_2) = (\Sigma_3, \hat{v})}{\varsigma, \Gamma \models (\Sigma_1, !e) \rightarrow (\Sigma_3, \hat{v}^\ell)} \quad \text{assign} \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \rightarrow (\Sigma_2, (\hat{R}, \hat{W})^\ell) \quad \varsigma, \Gamma \models (\Sigma_2, e_2) \rightarrow (\Sigma_3, \hat{v}) \quad \hat{W}(\varsigma \sqcup \ell, \Gamma, \Sigma_3 \hat{v}) = \Sigma_4}{\varsigma, \Gamma \models (\Sigma_1, e_1 := e_2) \rightarrow (\Sigma_4, \hat{v})} \\
 \\
 \text{lib} \frac{\text{lookupU}(\Gamma, \Sigma, x) = v \quad \text{lookupM}(\Gamma, \Sigma, x) = \gamma \quad v \uparrow_{\Gamma, \Sigma} \gamma = \hat{v}}{\varsigma, \Gamma \models (\Sigma, x_{lib}) \rightarrow (\Sigma, \hat{v})}
 \end{array}$$

Fig. 2. Selected labeled semantics

evaluated, using the newly created pointer along with the updated heap. When applying a function closure (**app**), the body of the function is executed in the program semantics, under the elevated context consisting of the current security context raised to the label of the function closure. Creation and application of library closures, $F : (\varsigma, \Gamma, \Sigma_1, v) \rightarrow (\Sigma_2, v)$, is analogous.

Safe implementation of marshaling of references requires the ability to trap and modify reads and writes in order to marshal the values passed by the interaction. For this reason, references are represented as pairs of functions, one function for reading the reference, $\hat{R} : (\varsigma, \Gamma, \Sigma_1) \rightarrow (\Sigma_2, \hat{v})$, and one function for updating the reference, $\hat{W} : (\varsigma, \Gamma, \Sigma_1, \hat{v}) \rightarrow \Sigma_2$. This allows us to marshal references by wrapping the read and the write functions in functions that perform the marshaling of the values at the time of interaction, similar to lazy marshaling of lists [17]. Most languages do not support the creation of functions that are triggered on interaction with values such as references or objects, which means they cannot support marshaling of first-class mutable state. A notable exception to this is JavaScript that allows methods to be tied to different aspects of object interaction via the use of Proxy objects [22].

Creation of references given a fresh pointer into the labeled heap is defined by `lread` and `lwrite` as follows.

$$\begin{aligned} \text{lread}(\hat{\rho}) &= \lambda(\varsigma, \Gamma, (\hat{\sigma}, \sigma, \ddot{\sigma})) . ((\hat{\sigma}, \sigma, \ddot{\sigma}), \hat{v}), \text{ where } \hat{v} = \hat{\sigma}[\hat{\rho}] \\ \text{lwrite}(\hat{\rho}) &= \lambda(\varsigma, \Gamma, (\hat{\sigma}_1, \sigma_1, \ddot{\sigma}_1), \hat{v}) . (\hat{\sigma}_2, \sigma_1, \ddot{\sigma}_1) \\ &\text{ where } v^\ell = \hat{\sigma}_1[\hat{\rho}], \varsigma \sqsubseteq \ell, \hat{\sigma}_2 = \hat{\sigma}_1[\hat{\rho} \mapsto \hat{v}^\varsigma] \end{aligned}$$

References (**ref**) are created by selecting a fresh heap location made to point to the value of the reference. The heap location is then used to create a pair of access functions. The created reference follows the same intuition as for all created values. All values are labeled L upon creation, which is why the pair of access functions are labeled L in **ref**. Note that the value that the reference is referring to may be labeled differently, due to the distinction between reference as a value and the value the reference is referring to. Dereferencing (**deref**) uses the read function of the reference to get the value to be read, while assignment (**assign**) uses the write function. Creation and use of library references, $R : (\varsigma, \Gamma, \Sigma_1) \rightarrow (\Sigma_2, v)$ and $W : (\varsigma, \Gamma, \Sigma_1, v) \rightarrow \Sigma_2$ is analogous.

It is worthwhile to point out the *no-sensitive upgrade* (NSU) check in `lwrite`, which demands that the context, which the label of the reference is a part of, is lower or equal to the label of the referenced value, $\varsigma \sqsubseteq \ell$. Allowing labels of values to change freely leads to an unsound system, due to the possibility of implicit flows into the labels themselves [1, 28].

Disregarding the encoding of functions and references into functions, up to this point, the labeled and unlabeled semantics are equivalent to their standard formulations. The essence of this paper is in the marshaling of values between the program and the library, performed by the unlabeled and relabeling functions, defined in the following section.

3.3 Marshaling

All interaction between the program and the library is initiated by lifting named library values into the program. This is done (**lib**) by looking up the value, and the corresponding relabel model used to relabel the value. Interaction with the relabeled value may cause further marshaling. Unlabeling of a value is done w.r.t. an unlabel model, φ , which defines how to store the removed label(s) in the model state. Relabeling of a value is done w.r.t. a relabel model, γ , which defines how to compute the label in terms of the model state. Formally, unlabeling is a function of the form $\hat{v} \downarrow_{\varsigma, \Gamma, \Sigma_1} \varphi = (\Sigma_2, v)$ taking a labeled value \hat{v} , an environment, $\varsigma, \Gamma, \Sigma_1$ and an unlabel model φ and returning an updated heap, Σ_2 , and an unlabeled value v . Similarly, relabeling is a function of the form $v \uparrow_{\Gamma, \Sigma} \gamma = \hat{v}$, taking an unlabeled value, v , an environment, Γ, Σ , and a relabel model, γ , and returning a labeled value \hat{v} . The only modified part of the heap for both unlabeling and relabeling is the model heap.

There are six types of values: integers, tuples, lists, records, higher-order functions and references. In the rest of this section we describe how to evaluate label terms (used when relabeling) and how to marshal higher-order functions

and references. We refer the reader to the full version of this paper [27] for the treatment of the other constructs.

Label Terms. Evaluation of label terms is done w.r.t. a model state, where `lookupM` is used to traverse the model scope chain to find the first label corresponding to a given label variable.

$$\begin{aligned} \llbracket \alpha \rrbracket_{\Gamma, \Sigma} &= \begin{cases} \ell, & \text{if } \text{lookupM}(\Gamma, \Sigma, \alpha) = \ell \\ L, & \text{otherwise} \end{cases} \\ \llbracket \ell \rrbracket_{\Gamma, \Sigma} &= \ell \\ \llbracket \kappa_1 \sqcup \kappa_2 \rrbracket_{\Gamma, \Sigma} &= \llbracket \kappa_1 \rrbracket_{\Gamma, \Sigma} \sqcup \llbracket \kappa_2 \rrbracket_{\Gamma, \Sigma} \end{aligned}$$

Higher-Order Functions. Marshaling of higher-order functions involves both marshaling the functions as values as well as ensuring the parameter and return value are properly marshaled.

Unlabeling. Unlabeling a program closure removes and stores the label and returns a library closure created by wrapping the program closure. The library closure is tied to the abstract name, π , used by the wrapper to relabel the parameters before the call and unlabel the result after the call.

$$\hat{F}^\ell \downarrow_{\zeta, \Gamma, \Sigma} \# \pi^\alpha = (\text{updateM}(\zeta, \Gamma, \Sigma, \alpha, \ell), \text{u-lclos}(\hat{F}, \ell, \# \pi))$$

The translation of a program closure, \hat{F} , into an library closure is performed by `u-lclos`, that takes the program closure, the label of the program closure and the abstract name. When the library closure returned by `u-lclos` is applied the following occurs. First, the function call model bound to the abstract name is fetched using `lookupM`. The function call model contains a label term representing the security context of the application, how to relabel the parameter and how to unlabel the return value. Second, the relabel model, γ , is used to relabel the parameter, v_1 . Third, the program closure is called in the security context of the call raised to the label of the closure and the evaluation of the context label term, κ . The result of the call is a labeled value, \hat{v}_2 . Finally, \hat{v}_2 is unlabeled which gives the result, v_2 , of the application of the unlabeled closure. Notice that all relabeling and unlabeling is done with respect to the model state of the caller.

$$\begin{aligned} \text{u-lclos}(\hat{F}, \ell_1, \# \pi) &= \lambda(\zeta, \Gamma, \Sigma_1, v_1) . (\Sigma_3, v_2) \\ \text{where } \kappa \vdash \gamma \rightarrow \varphi &= \text{lookupM}(\Gamma, \Sigma_1, \pi) \\ \ell_2 &= \llbracket \kappa \rrbracket_{\Gamma, \Sigma_1} \\ \hat{v}_1 &= v_1 \uparrow_{\Gamma, \Sigma_1} \gamma \\ (\Sigma_2, \hat{v}_2) &= \hat{F}(\zeta \sqcup \ell_1 \sqcup \ell_2, \Gamma, \Sigma_1, \hat{v}_1) \\ (\Sigma_3, v_2) &= \hat{v}_2 \downarrow_{\zeta \sqcup \ell_1 \sqcup \ell_2, \Gamma, \Sigma_2} \varphi \end{aligned}$$

Relabeling. Relabeling a library closure is done by labeling the program closure created by wrapping the library closure. The wrapper unlabels the arguments before the call and relabels the result of the call.

$$F \uparrow_{\Sigma, (\underline{\rho}, \underline{\rho}, \underline{\rho})} (\varphi \rightarrow \gamma, \underline{\zeta})^\kappa = \text{l-uclos}(F, \underline{\rho}, (\varphi \rightarrow \gamma, \underline{\zeta}))^{\llbracket \kappa \rrbracket_{(\underline{\rho}, \underline{\rho}, \underline{\rho}), \Sigma}}$$

The process is controlled by the function relabel model, $(\varphi \rightarrow \gamma, \underline{\zeta})^\kappa$, where the evaluation of κ gives the label of the wrapper closure.

The translation of the library closure, F , into a program closure is performed by l-uclos , which takes the library closure, the current model frame stack, the unlabel model for the parameters, φ , the relabel model for the return value, γ , and the effect constraints, ζ . When called the

$$\begin{aligned} \text{l-uclos}(F, \hat{\rho}_2, (\varphi \rightarrow \gamma, \zeta)) = & \\ & \lambda(\varsigma, (\hat{\rho}, \underline{\rho}, \hat{\rho}_1), (\hat{\sigma}, \sigma, \hat{\sigma}), \hat{v}_1). (\Sigma_4, \hat{v}_2) \\ \text{where } \Sigma_1 = & (\hat{\sigma}, \sigma, \hat{\sigma}[\hat{\rho} \mapsto \emptyset]), \hat{\rho} \text{ fresh} \\ (\Sigma_2, v_1) = & \hat{v}_1 \downarrow_{\varsigma, (\hat{\rho}, \underline{\rho}, \hat{\rho}_2), \Sigma_1} \varphi \\ \Sigma_3 = & \{ \{ \zeta \} \}_{\varsigma, (\hat{\rho}, \underline{\rho}, \hat{\rho}_2), \Sigma_2} \\ (\Sigma_4, v_2) = & F(\varsigma, (\hat{\rho}, \underline{\rho}, \hat{\rho}_2), \Sigma_3, v_1) \\ \hat{v}_2 = & v_2 \uparrow_{(\hat{\rho}, \underline{\rho}, \hat{\rho}_2), \Sigma_4} \gamma \end{aligned}$$

program closure produces a fresh frame pointer, pointing to a new model frame in the model heap. The parameter to the library function is unlabeled based on the unlabel model, φ , and the effect constraints, ζ , are evaluated to update the model state accordingly. After that, the library function is called with the unlabeled parameter in the security context, ς , of the call. The result of the function call is relabeled with the relabel model, γ , and returned to the program. Note that all labeling and unlabeling is done w.r.t. the model frame stack of the unlabeled closure. Also note that the order is important; if the unlabeling of the parameter occurs after evaluating the effect constraints, the label of the parameter cannot be used when updating the model state with the side effects.

Effect Constraints. Effect constraints define how a library function interacts with unlabeled program functions and references and how the library function changes the model state. Model state changes are effectuated on call to the library function whereas effect constraints that define interaction with unlabeled program functions and references are stored in the model state. When a library function or reference is interacted with, the abstract name will tie the interaction to the corresponding effect constraint in the model state of the interaction. The meaning of the effect constraints is defined as follows

$$\begin{aligned} \{ \{ \# \alpha \rightarrow \varphi \} \}_{\varsigma, \Gamma, \Sigma} &= \text{defineM}(\Gamma, \Sigma, \alpha, \varphi) \\ \{ \{ \kappa \vdash \# \alpha \leftarrow \gamma \} \}_{\varsigma, \Gamma, \Sigma} &= \text{defineM}(\Gamma, \Sigma, \alpha, \kappa \vdash \gamma) \\ \{ \{ \kappa \vdash \# \alpha \gamma \rightarrow \varphi \} \}_{\varsigma, \Gamma, \Sigma} &= \text{defineM}(\Gamma, \Sigma, \alpha, \kappa \vdash \gamma \rightarrow \varphi) \\ \{ \{ \kappa_1 \vdash \alpha \leftarrow \kappa_2 \} \}_{\varsigma, \Gamma, \Sigma} &= \text{updateM}(\varsigma \sqcup \llbracket \kappa_1 \rrbracket_{\Gamma, \Sigma}, \Gamma, \Sigma, \alpha, \llbracket \kappa_2 \rrbracket_{\Sigma, \Gamma}) \\ &\text{when } \varsigma \sqcup \llbracket \kappa_1 \rrbracket_{\Gamma, \Sigma} \sqsubseteq \text{lookupM}(\Gamma, \Sigma, \alpha) \end{aligned}$$

where defineM binds the name α to its corresponding model value in the top model frame, if α is not defined in that model frame, updateM updates the label pointed to by α in the scope chain, or inserts it if it is not present, and lookupM returns the model value that is the first to match the name α in the scope chain.

References. Marshaling of references shares some similarities with marshaling of higher-order functions. Calling a function passes the argument and the return value in opposite directions, similar to reading and writing to a reference.

Unlabeling. Unlabeling a program reference removes and stores the label, and the read and write functions are wrapped to create library counterparts.

$$(\hat{R}, \hat{W})^\ell \downarrow_{\varsigma, \Gamma, \Sigma} \# \pi^\alpha = \text{(updateM}(\varsigma, \Gamma, \Sigma, \alpha, \ell), \text{(u-lread}(\hat{R}, \ell, \# \pi), \text{u-lwrite}(\hat{W}, \ell, \# \pi)))$$

The read and the write functions are translated independently w.r.t. the abstract name $\# \pi$.

The program read function, \hat{R} is translated by `u-lread`, which takes the read function, the label of the reference and the abstract name. When the resulting library read function is interacted with,

$$\begin{aligned} \text{u-lread}(\hat{R}, \ell, \# \pi) &= \lambda(\varsigma, \Gamma, \Sigma_1) . (\Sigma_3, v) \\ \text{where } \varphi &= \text{lookupM}(\Gamma, \Sigma_1, \pi) \\ (\Sigma_2, \hat{v}) &= \hat{R}(\varsigma \sqcup \ell, \Gamma, \Sigma_1) \\ (\Sigma_3, v) &= \hat{v} \downarrow_{\varsigma \sqcup \ell, \Gamma, \Sigma_2} \varphi \end{aligned}$$

the program read function is used to get the labeled value of the reference. This value must be unlabeled before being returned, which is done by looking up a program reference read model, φ , in the model state of the interaction. It is the model of the caller, i.e., a library function model that provides the read model for the references it reads.

The program write function, \hat{W} is translated by `u-lwrite`, which takes the write function, the label of the reference and the abstract name. When the resulting library write function is used, the associ-

$$\begin{aligned} \text{u-lwrite}(\hat{W}, \ell, \# \pi) &= \lambda(\varsigma, \Gamma, \Sigma_1, v) . \Sigma_2 \\ \text{where } \kappa \vdash \gamma &= \text{lookupM}(\Gamma, \Sigma_1, \pi) \\ \hat{v} &= v \uparrow_{\Gamma, \Sigma_1} \gamma \\ \Sigma_2 &= \hat{W}(\varsigma \sqcup \ell \sqcup \llbracket \kappa \rrbracket_{\Gamma, \Sigma_1}, \Gamma, \Sigma, \hat{v}) \end{aligned}$$

ated program reference write model, $\kappa \vdash \gamma$, is fetched in the current model state. This model defines both how to relabel the written unlabeled value, and the context in which the write occurs. Then the unlabeled value, v is relabeled before being written using the labeled write function in a context consisting of the current security context of the call raised to the reference label and the evaluation of the context label term, κ .

Relabeling. Relabeling a library reference is done by translating the read and write functions into program counterparts and relabeling the result.

$$(R, W) \uparrow_{\Sigma, (\hat{\rho}, \rho, \hat{\rho})} \text{ref}(\varphi, \gamma)^\kappa = (\text{l-uread}(R, \hat{\rho}, \gamma), \text{l-uwrite}(W, \hat{\rho}, \gamma, \varphi))^{\llbracket \kappa \rrbracket_{(\hat{\rho}, \rho, \hat{\rho}), \Sigma}}$$

The read and the write functions are translated independently w.r.t. the relabel model, $\text{ref}(\varphi, \gamma)^\kappa$.

The library read function, R , is translated by `l-uread`, which takes the read function, the current model frame stack, and the relabel model, γ . When the resulting program read function is interacted with,

$$\begin{aligned} \text{l-uread}(R, \hat{\rho}, \gamma) &= \lambda(\varsigma, (\hat{\rho}, \rho, \hat{\rho}_1), \Sigma_1) . (\Sigma_2, \hat{v}) \\ \text{where } (\Sigma_2, v) &= R(\varsigma, (\hat{\rho}, \rho, \hat{\rho}_1), \Sigma_1) \\ \hat{v} &= v \uparrow_{(\hat{\rho}, \rho, \hat{\rho}_2), \Sigma_2} \gamma \end{aligned}$$

the unlabeled read function is used to fetch the unlabeled value of the reference. The result is relabeled using the relabel model in the model state of the reference and the result is returned.

The library write function W is translated by $l\text{-uwrite}$, which takes the write function, the current model frame stack, the relabel model, γ , and the unlabel model, φ . The reason $l\text{-uwrite}$ takes the relabel model in addition to the unlabel model is that it is used to calculate the label against which the NSU check is made. The label of the stored value is represented by the label term of the relabel model, extracted by the $l\text{blterm}$ function, defined in the obvious way by pattern matching. If the write is allowed, the labeled value to be written to the library reference is raised to the context ς , before being unlabeled using the unlabel model, φ . Finally, the unlabeled value is written to the library reference, using the unlabeled write function.

$$l\text{-uwrite}(W, \underline{\rho}_2, \gamma, \varphi) = \lambda(\varsigma, (\hat{\rho}, \rho, \underline{\rho}_1), \Sigma_1, \hat{v}) . \Sigma_3$$

$$\text{where } \ell = \llbracket l\text{blterm}(\gamma) \rrbracket_{(\hat{\rho}, \rho, \underline{\rho}_2), \Sigma_1}, \varsigma \sqsubseteq \ell$$

$$(\Sigma_2, v) = \hat{v}^\varsigma \downarrow_{\varsigma, (\hat{\rho}, \rho, \underline{\rho}_2), \Sigma_1} \varphi$$

$$\Sigma_3 = W(\varsigma, (\hat{\rho}, \rho, \underline{\rho}_1), \Sigma_2, v)$$

The program calls the library function f , which takes a parameter, and creates a reference r initially set to the value of the parameter. f returns a pair, where the first element is a function that, given any argument, will dereference the reference and the second element is the actual reference. This pair is stored as (g, r) . Thereafter, r is assigned the value 15^H , before g is called with the parameter 20^L .

Interaction with the Model Heap. To see how higher-order functions and references interact with the model heap, consider the code snippet below to the right. The program calls the library function f , which takes a parameter, and creates a reference r initially set to the value of the parameter. f returns a pair, where the first element is a function that, given any argument, will dereference the reference and the second element is the actual reference. This pair is stored as (g, r) . Thereafter, r is assigned the value 15^H , before g is called with the parameter 20^L .

```
let (g, r) = lib f 10
in r := upg 15 H;
  g 20
%%
lbl l :: L
mod r :: ref (1, 1)
mod f :: x -> (y -> l, r)
fun f x = let r = ref x
          in (\y. !r, r)
```

The following occurs w.r.t. relabeling and unlabeled in the program, where the initial setting can be seen in Fig. 3.

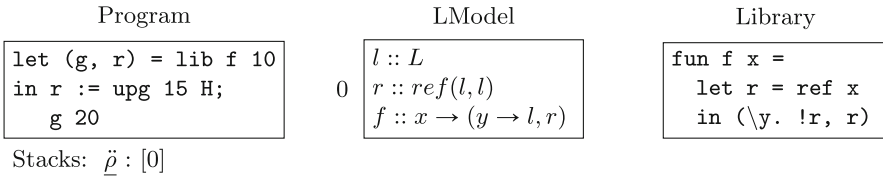


Fig. 3. Initial structure

When f is lifted to the program, $l\text{-uclos}$ is used to relabel the library closure, which will copy the model frame stack to the wrapped f and store the function model $x \rightarrow (y \rightarrow l, r)$. In the example, the resulting program closure is applied to 10^L , which causes a new model frame to be allocated on the model heap, into which the argument is unlabeled, causing L to be stored in the new model frame as the label for x , and the pointer to the new model frame is stored in the model frame stack. After this, the actual unlabeled function is called, which

results in the returned pair being relabeled. The relabeling of the pair results in `l-uclos` being used to relabel `\y. !r` with the model $y \rightarrow l$, and `l-uread` and `l-uwrite` being used to relabel `r` with the reference model r . The key here is that the relabeling occurs in the same model state, which means that the produced program function and reference will be bound to the same model frame stack. This causes writes to the reference to modify the model frame shared with the function, ensuring that they have the same view of the model of the reference. The entire process is highlighted in Fig. 4.

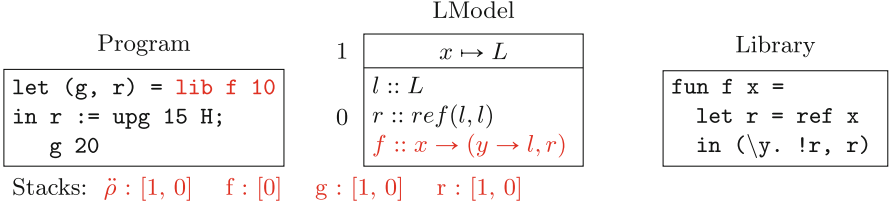


Fig. 4. Calling relabeled closure

When the program writes to the reference (`r := upg 15 H`), the closure from `l-uwrite` is triggered, causing l in the shared model frame to be updated to H , which can be seen in Fig. 5. Note that the pointer to the model frame created from the call to the wrapped `f` is removed from the model frame stack. This ensures any subsequent calls to the wrapped `f`, as well as any created wrappers will not be able to use that model frame, as it belongs only to the first call to the wrapped `f` and the created wrappers *within* the call. When the function `g` is called, it will trigger its `l-uclos` wrapper and, as can be seen in Fig. 6, the model $y \rightarrow l$ is used in the `l-uclos` wrapper for `g`, with l being used to relabel the result. Since l was modified by the writing to the reference (Fig. 5), the shared view of the library model state, will make the function `g` return a secret value.

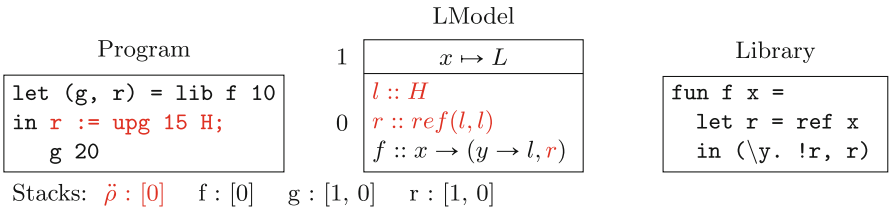
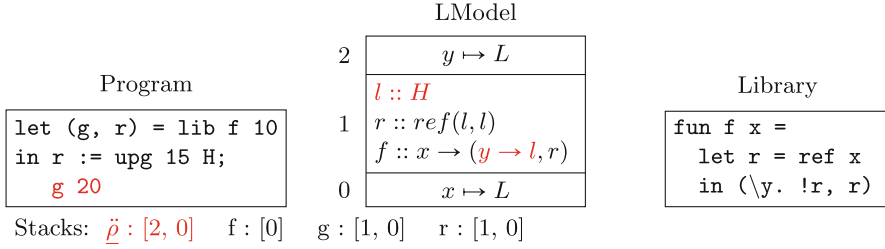


Fig. 5. Writing to `r`

Fig. 6. Calling *g*

4 Examples

In the following section we provide some examples to highlight how the language would interact with common programming techniques. The language used in this section is an extended version of the language of the paper. The major differences are the addition of records, functions with multiple arguments, a limited form of pattern matching, and optional unlabeled. The extensions are all present as experimental features in the implementation. In all examples, the code above %% is the program and the code below is the library.

Writebacks. Returning two or more results from a function can be done in two ways: (1) tupling the result, or (2) by using writebacks. When using writebacks for, e.g., reading a file, the read function is provided a pointer to a place in the memory where the contents of the file should be stored instead of returning a pointer to the data.

In our language, writebacks can be modeled by passing program references to the library as shown to the right. In the example, the program variable `buf` is a program reference. The reference is passed to the library function `action` that writes the result to the buffer. When interacting with a program reference, the reference is given an abstract name (`b` for buffer in this case) that the function interacting with the buffer uses to relabel the interaction.

In case the function used the writeback under secret control, represented by the model `mod action :: #b -> L { | H | - #b <- H | }`, the example would fail due to NSU. The reason being the value the reference `buf` is pointing to is public, and is not allowed to change label under secret control. Modifying the declaration of `buf` to be `let buf = ref (upg 0 H)` solves this, as the reference will point to a secret value.

```
let buf = ref 0
fun main () = (lib action) buf;
!buf
%%
let data = 42
mod action :: #b -> L { | #b <- H | }
fun action b = b := data; ()
```

Library State. Libraries often keep state, e.g., error codes, computation results or options set by the program. Typical examples are the predefined object properties `$1, ..., $9` from JavaScript RegExp [23].

The example to the right shows how state can be used to store error information. In the example, the function `action` may fail depending on the value of the

```
fun main () =
  (lib action) (upg 42 H);
  print !(lib errno)
%%
lbl l :: L
mod errno :: ref (1, 1)
let errno = ref 0

mod action :: a -> L {| l <- a |}
fun action x = if x == 1
  then errno := 1
  else ();
  ()
```

parameter. The reason it failed is stored into the library reference `errno`, which is modeled by a security label used to relabel program reads and writes of the reference. Since the update of `errno` is conditional, it means that the value of `errno` is dependent of the argument of the `action` function. To model this, the argument label is stored in the model variable `a`, which is used to update the security label of `errno`. Note that the update of the security label is independent on whether the operation fails or not. This is needed to ensure that the label of `errno` is independent of secrets. The label of `errno` indicates that the error code is public. Consider the case where an action sets the error code under secret control, represented by the following model `mod action :: a -> L {| H |- l <- a |}`. If such an action was used our system would halt execution, since the update of the error code would trigger NSU.

The One-Place Buffer. In the previous example, the library state is exposed to the program, which can freely read and write to `errno`. Frequently it is good practice to hide the internal state of the library and only allow the program to access it indirectly via the functions of the library. We exemplify this by implementing a simple one-place buffer, seen to the right. While simple, the example captures the essence of, e.g., buffered file access.

```
fun main () = (lib set) (upg 42 H);
              (lib getAsync) print;
              (lib get) ()
%%
lbl l :: L
let buf = ref 0

mod set :: a -> a {| l <- a |}
fun set x = buf := x

mod get :: _ -> l
fun get () = !buf

mod getAsync :: #cb -> L {| #cb l -> _ |}
fun getAsync cb = cb !buf; ()
```

Since there is no model for `buf`, it is not accessible from the program. Instead, the state of the library is modeled using the label `l`. This label is used by the operations that give the program access to the buffer contents. When setting the value of the buffer via `set`, the label of the value is used to update the label of the library state. When reading, either via the synchronous function `get` or via the asynchronous function `getAsync`, `l` is used to relabel the dereferenced value from `buf`. In the synchronous case by relabeling the dereferenced value directly, and in the asynchronous case by relabeling the parameter to the callback. Note the use of the wildcard `_` to indicate values that are not important for the model.

Stored Callbacks. Stored callbacks are callbacks that are saved in the internal state of the library and used, e.g., to signal the occurrence of some event. A typical example of stored callbacks is the event handlers present in many languages.

Consider the program to the right that registers an event handler by storing the event handler (`print` in this case) in the `event` reference of the library. The relabel model of the `event` will unlabel the function and give it the abstract name `event`.

```
fun main () = lib event := print;
                    (lib fire) (upg 42 H)
%%
lbl 1 :: L
mod event :: ref (1, #event^1)
let event = ref 0

mod fire :: a -> L { | #event a -> _ | }
fun fire x = !event x; ()
```

The event is triggered by calling the `fire` function, which takes the event data and passes it to the stored event handler. In the example, the `fire` function may be called from the program. In a practical setting, events may be triggered by interacting with the library (e.g., by adding values to a data structure) or from the library itself to indicate that certain events, such as mouse movement or clicks, have occurred.

In the example, it is not possible to fetch the event handler from the library and call it. In order to allow for this, we have to change the relabel model for the library reference to relabel read interactions as functions, changing the `event` model to be `mod event :: ref (a -> b { | #event a -> b | }^1, #event^1)`. To understand the new relabel model we must recognize that unlabeled program functions that are passed back need to be relabeled as any other library function. In this case, the library function that should be relabeled calls the unlabeled program function, and needs a corresponding call model. The result is a function that unlabels its argument into the label variable `a`, which is used to relabel the argument before calling the program function. The result of calling the program function is unlabeled into the label variable `b`, which in turn is used to relabel the result of the relabeled function.

5 Case Study

For case study, we model an API inspired by the `fs` API of `node.js` [10]. In the interest of exposition we model the file system state as a single label as shown to the right. The extension of the model to nested records is simple but space demanding.

```
lbl 1 :: L
mod state :: ref (1, 1)
let state = ref 1
```

Examples of functions in the API are the `rmdir` function and its synchronous sibling `rmdirSync`. Both will, given a path, remove the folder pointed to by the path. In addition, `rmdir` also takes a callback that is called with an error if the removal of the folder pointed to by the path fails.

```
mod rmdirSync :: a -> 1 + a { | 1 <- a | }
mod rmdir :: (a, #cb) -> L { | 1 <- a, #cb (1 + a) -> b | }
```

We use the name `a` to represent the path and the abstract name `cb` to represent the callback. From a modeling standpoint, we need to ensure that the level of

the path is propagated to the state, since removing the folder influences the file system state. We can see this in the effect constraint $l \leftarrow a$, where the label of the path is propagated to the label of the state. The success of the operation is depending on the library state and the security label of the path, $l + a$. Where `rmdirSync` returns the result, `rmdir` communicates the result to the callback as an argument, `#cb (l + a)`. The immediate return value of the latter is `undefined`, regardless of the outcome of the operation and hence labeled `L`.

A more complex function in the API is `createWriteStream` that returns a record. Calling `createWriteStream` with a path and an optional argument that defines options (e.g. the encoding) returns a `WriteStream`.

```
mod createWriteStream :: (a, b)
-> { path      : a
    , bytesWritten : a + b + 1
    , open      : #op -> L {l | #op (a + b + 1) -> o |}
    , close     : #cl -> L {l | #cl (a + b + 1) -> c |}
}
```

The `WriteStream` has four parts; the fields `path` and `bytesWritten`, as well as the events `open` and `close`. For the model of the returned record, the property `path` is modeled by the argument `a`, which is the label of the path. The property `bytesWritten`, which corresponds to the amount of bytes written so far, is modeled as the least upper bound of `a`, `b` and `1`, i.e., the path, the options and the current library state. The events are modeled as functions that accept (and store) callbacks—the event handler—as modeled by the properties `open` and `close`. When the stream is opened or closed, the path, the options and the current library state all influence the parameter to those callbacks.

To contrast the case study with the examples, note that Sect. 4 makes the assumption that the source code of the library is available (albeit not supporting the labeled semantics) whereas this section makes the assumption it is not. Both cases are common, and can be modeled in our approach. In case the source code is indeed available an interesting line of future work is to look at the possibilities of automatically deducing models, e.g., using something similar to summary functions [26].

6 Correctness

The correctness of the language is complicated by the fact that it is parameterized over a library model that defines how to marshal values between the program and the library. Since we make no assumption on the implementation language of the library or the availability of the source code we cannot reason about the correctness of the model w.r.t. the library. Instead we assume the correctness of library models in terms of three hypotheses used in the noninterference proof. The low-equivalence definition, the model hypotheses and more information on the proof can be found in the full version of the paper [27].

We prove noninterference assuming that the library model correctly models the library as the preservation of a low-equivalence relation under execution.

Apart from covering a larger language, the proof improves over [17] in two important aspects: (1) it significantly weakens the model hypothesis, and (2) the proof has been formalized in Coq [19].

Theorem 1. (Noninterference of labeled execution)

$$(\Gamma, \Sigma_1) \simeq (\Gamma', \Sigma'_1) \wedge \varsigma, \Gamma \models (\Sigma_1, e) \rightarrow (\Sigma_2, \hat{v}) \wedge \\ \varsigma, \Gamma' \models (\Sigma'_1, e) \rightarrow (\Sigma'_2, \hat{v}') \Rightarrow (\Gamma, \Sigma_2) \simeq (\Gamma', \Sigma'_2) \wedge \hat{v} \simeq \hat{v}'$$

Proof. By mutual induction on labeled and unlabeled evaluation (via `u-lclos` and `l-uclos`). The theorem makes use of *confinement*, i.e., that evaluation under high security does not modify the public part of the environment.

7 Related Work

Bielova and Rezk present a comprehensive taxonomy of information flow monitors [4]. Some monitors [3, 14–16] and secure multi-execution [6, 12, 13, 20, 24] mechanisms have been integrated in a browser. Bichhawat et al. instrumented the WebKit JavaScript interpreter [3]. While taking advantage of the current optimizations in the interpreter, it loses the differentiation between the program and library execution. FlowFox [13], which implements *secure multi-execution (SME)* [6], modifies the SpiderMonkey engine in two ways: (1) augmenting the internal objects representing the JavaScript context with a current execution level, as well as a boolean indicating if SME is active, and (2) augmenting the internal representation of JavaScript values with a security level. Unfortunately, API calls are only treated as I/O actions. JSFlow [16] is an information-flow aware JavaScript interpreter, augmented with security labels on the JavaScript values. In order to allow for libraries in JSFlow, deep hand-written models must be used, with reimplementations of the libraries as a result [15]. To allow for scaling, JSFlow attempts to automatically wrap libraries, albeit in an ad-hoc manner. While the correctness of simple examples are easy to see, the correctness and scalability when passing, e.g., functions to and from the library remain unclear. Bauer et al. [2] developed a light-weight coarse-grained run-time monitor for Chromium, using taint tracking, to help reasoning about information flow in a fully fledged browser. In this work, formal models of, e.g., cookies, history and the *document object model (DOM)* are defined, as well as event handlers, to model the browser internals and help prove noninterference. Heule et al. [18] provided a theoretical foundation for a language-based approach for coarse-grained dynamic information flow control, that can be applied to any programming language where external effects can be controlled. A first step for handling libraries in environments where dynamic information flow control is not possible was taken by Hedin et al. [17], falling short by not supporting references, and thereby not allowing for first-class mutable state in combination with higher-order functions.

Findler and Feleisen’s higher-order contracts [9] address the problem of checking contracts at the boundary between statically type-checked and dynamically type-checked code. The problem relates to the problem of interfacing with

libraries where it is impossible to check dynamic information flow control. In particular, when considering function values crossing the boundary, the compliance of such function values with their respective contracts is undecidable. Findler and Feleisen proposed to wrap the function and check the contract at the point where the function is called. This is comparable to how we handle structured data, including references and function values. A question for future work is if we can remove our abstract identifiers for function values and references, and instead inject the unlabeled/relabeling functionality using proxies, similar to how it is done in higher-order contract checking [8]. If a contract is violated, the proper assignment of blame must be given [7, 11]. In static information flow checking, the assignment of blame has been investigated by King et al. for information flow violations [21]. Although our work can be seen as an application of dynamic higher-order contract checking for information flow contracts, we do not consider assigning blame. Indeed, runtime detection of a library which does not obey the specified contract (i.e. the given model) is not possible in this work.

8 Conclusion

Based on a central idea of a model heap, we have developed a foundation for information flow tracking in the presence of libraries with side effects in a language with higher-order functions, first-class state and lazy-marshaling—three cornerstones of practical libraries. We have implemented a prototype to verify the examples and performed a larger case study that shows that the language is able to model key parts of a real file system library. In addition, we have formalized the language and its correctness proof in Coq.

Future work includes support for model abstraction and application, and dependent models. Thanks to the three cornerstones, we believe modeling JavaScript objects does not require development of new theory, indicating that it is possible to use this technique in tools like JSFlow.

Acknowledgments. This work was partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

References

1. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: PLAS (2009)
2. Bauer, L., Cai, S., Jia, L., Passaro, T., Stroucken, M., Tian, Y.: Run-time monitoring and formal analysis of information flows in chromium. In: NDSS. The Internet Society (2015)
3. Bichhawat, A., Rajani, V., Garg, D., Hammer, C.: Information flow control in WebKit’s JavaScript bytecode. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 159–178. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_9
4. Bielova, N., Rezk, T.: A taxonomy of information flow monitors. In: Piessens, F., Viganò, L. (eds.) POST 2016. LNCS, vol. 9635, pp. 46–67. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49635-0_3

5. Denning, D.E.: A lattice model of secure information flow. *Commun. ACM* **19**(5), 236–243 (1976)
6. Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: *S&P* (2010)
7. Dimoulas, C., Findler, R.B., Flanagan, C., Felleisen, M.: Correct blame for contracts: no more scapegoating. In: *POPL* (2011)
8. Dimoulas, C., New, M.S., Findler, R.B., Felleisen, M.: Oh Lord, please don't let contracts be misunderstood (functional pearl). In: *ICFP* (2016)
9. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: *ICFP* (2002)
10. File System–Node.js v9.2.0 Documentation. <https://nodejs.org/api/fs.html>. Accessed Nov 2017
11. Greenberg, M., Pierce, B.C., Weirich, S.: Contracts made manifest. In: *POPL* (2010)
12. Groef, W.D., Devriese, D., Nikiforakis, N., Piessens, F.: FlowFox: a web browser with flexible and precise information flow control. In: *CCS* (2012)
13. Groef, W.D., Devriese, D., Nikiforakis, N., Piessens, F.: Secure multi-execution of web scripts: theory and practice. *J. Comput. Secur.* **22**, 469–509 (2014)
14. Hedin, D., Bello, L., Sabelfeld, A.: Information-flow security for JavaScript and its APIs. *J. Comput. Secur.* **24**, 181–234 (2015)
15. Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: JSFlow: tracking information flow in JavaScript and its APIs. In: *SAC* (2014)
16. Hedin, D., Sabelfeld, A.: Information-flow security for a core of JavaScript. In: *CSF* (2012)
17. Hedin, D., Sjösten, A., Piessens, F., Sabelfeld, A.: A principled approach to tracking information flow in the presence of libraries. In: Maffei, M., Ryan, M. (eds.) *POST 2017*. LNCS, vol. 10204, pp. 49–70. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_3
18. Heule, S., Stefan, D., Yang, E.Z., Mitchell, J.C., Russo, A.: IFC inside: retrofitting languages with dynamic information flow control. In: Focardi, R., Myers, A. (eds.) *POST 2015*. LNCS, vol. 9036, pp. 11–31. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46666-7_2
19. INRIA: The Coq Proof Assistant. <https://coq.inria.fr/>. Accessed Nov 2017
20. Kashyap, V., Wiedermann, B., Hardekopf, B.: Timing- and termination-sensitive secure information flow: exploring a new approach. In: *S&P* (2011)
21. King, D., Jaeger, T., Jha, S., Seshia, S.A.: Effective blame for information-flow violations. In: *FSE* (2008)
22. Mozilla Developer Network: Proxy. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy. Accessed Mar 2018
23. Mozilla Developer Network: RegExp. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp. Accessed Mar 2018
24. Rafnsson, W., Sabelfeld, A.: Secure multi-execution: fine-grained, declassification-aware, and transparent. In: *CSF* (2013)
25. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. *IEEE J. Sel. Areas Commun.* **21**(1), 5–19 (2003)
26. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications* (1981)
27. Sjösten, A., Hedin, D., Sabelfeld, A.: Information Flow Tracking for Side-effectful Libraries - Full version. <http://www.cse.chalmers.se/research/group/security/side-effectful-libraries/>
28. Zdancewic, S.A.: Programming languages for information security. Ph.D. thesis, Cornell University (2002)



On a Verification Framework for Certifying Distributed Algorithms: Distributed Checking and Consistency

Kim Völlinger^(✉) and Samira Akili

Humboldt University of Berlin, Berlin, Germany
voellinger@hu-berlin.de

Abstract. A major problem in software engineering is assuring the correctness of a distributed system. A *certifying distributed algorithm* (CDA) computes for its input-output pair (i, o) an additional *witness* w – a formal argument for the correctness of (i, o) . Each CDA features a *witness predicate* such that if the witness predicate holds for a triple (i, o, w) , the input-output pair (i, o) is correct. An accompanying *checker* algorithm decides the witness predicate. Consequently, a user of a CDA does not have to trust the CDA but its checker algorithm. Usually, a checker is simpler and its verification is feasible. To sum up, the idea of a CDA is to adapt the underlying algorithm of a program at design-time such that it verifies its own output at runtime. While certifying *sequential* algorithms are well-established, there are open questions on how to apply certification to *distributed algorithms*. In this paper, we discuss *distributed checking* of a *distributed witness*; one challenge is that all parts of a distributed witness have to be *consistent* with each other. Furthermore, we present a method for *formal instance verification* (i.e. obtaining a *machine-checked* proof that a particular input-output pair is correct), and implement the method in a framework for the theorem prover COQ.

Keywords: Certification · Distributed algorithm
Formal instance verification

1 Introduction

A major problem in software engineering is assuring the *correctness* of distributed systems. A distributed system consist of computing components that can communicate with each other. An algorithm that is designed to run on a distributed system is called a distributed algorithm. The correctness of a distributed algorithm usually relies on subtle arguments in hand-written proofs. Consequently, these proofs can easily be flawed. While complete formal verification is often too costly, testing is not sufficient if the system is of critical importance. Runtime verification tries to bridge this gap by being less costly than complete verification while still using mathematical reasoning.

We investigate certifying distributed algorithms. A *certifying distributed algorithm* (CDA) computes for its input-output pair (i, o) additionally a *witness* w – a formal argument for the correctness of the input-output pair (i, o) . Each CDA features a *witness predicate* such that if the witness predicate holds for a triple (i, o, w) , the input-output pair (i, o) is correct. A “correct” CDA always computes a witness such that the witness predicate holds. However, the idea is that a user of a CDA does not have to trust the algorithm. That is why, an accompanying *checker* algorithm decides the witness predicate at runtime. The user of a CDA has to trust neither the implementation nor the algorithm nor the execution. However, the user has to trust the checker to be sure that if the checker accepts on (i, o, w) , the particular input-output pair (i, o) is correct. Usually, a checker is simple and its verification feasible. By combining a CDA with program verification (e.g. verifying the checker), we gain formal instance correctness (i.e. a *machine-checked proof* that a particular input-output pair is correct). To sum up, the idea of a CDA is to adapt the underlying algorithm of a program at design-time such that it verifies its input-output pair at runtime. Hence, using a CDA is a formal method and a runtime verification technique.

While certifying *sequential* algorithms are well-established [19], there are open questions on how to apply certification to *distributed algorithms* [29]. In particular, there are various ways of applying the concept of certification to distributed algorithms. For instance, one question is whether to verify the input-output pair of a component or the distributed input-output pair of the system. Another question is whether the witness is checked by a distributed or sequential checker.

In this paper, we introduce a class of CDAs which features *distributed* checking of a *distributed* witness that verifies the correctness of a distributed input-output pair. Particularly, we discuss the challenge that all parts of a distributed witness have to be *consistent* with each other (Sect. 2). Moreover, we present a method for formal instance verification where we integrate the notion of consistency. We implement the method in a framework for the theorem prover COQ such that a verified distributed checker can be deployed on a real distributed system (Sect. 3). Our COQ formalization is on GITHUB¹. Moreover, we discuss related work (Sect. 4), as well as our contributions and future work (Sect. 5).

2 Certifying Terminating Distributed Algorithms

A distributed algorithm is designed to run on a distributed system, e.g. a network. We assume networks that are *asynchronous*, *static* and *id-based*. We model the topology of a network as a connected undirected graph $G = (V, E)$ with $V = \{1, 2, \dots, n\}$: a vertex represents a component and an edge a channel. A *distributed algorithm* consists of an algorithm for each component such that all components together solve one problem (e.g. leader election or coloring) [17, 25]. Components communicate with each other by sending messages via the channels.

¹ <https://github.com/voellinger/verified-certifying-distributed-algorithms/tree/master/Framework>.

A distributed algorithm can be either designed to terminate or to run continuously (e.g. a communication protocol). In this paper, we focus on *terminating* distributed algorithms. Thus, we deal with verifying a *distributed* input-output pair. In contrast, for a non-terminating algorithm, we would verify a behavior during the execution.

The rest of this Section is organized as follows. We start by defining the interface of a CDA. (Sect. 2.1). Moreover, we give a small example of a CDA to illustrate our formalization (Sect. 2.2). Subsequently, we define a witness predicate (Sect. 2.3) and a consistent witness (Sect. 2.4). For distributed checking of the witness predicate, we discuss how to decide a set of predicates for each component (Sect. 2.5). Finally, we define a class of CDAs (Sect. 2.6) and present the accompanying distributed checker of such a CDA (Sect. 2.7).

2.1 Interface of a CDA

The input of a distributed algorithm is *distributed* over the network in the way that each component gets a part of it. A *terminating* distributed algorithm computes an output in the way that each component computes a part of it. We call the algorithm of a component a *sub-algorithm* of the distributed algorithm, and a component's part of the (distributed) input/output its *sub-input/sub-output*. As usual when considering distributed algorithms, we abstract from distributing the input and collecting the sub-output.

Analogously to the computation of the output, a CDA additionally computes a distributed witness. We then call the algorithm of a component a *certifying* sub-algorithm of the CDA, and a component's part of the witness its *sub-witness*. We distinguish between a witness and a *potential* witness. While a witness is a proper correctness argument, a potential witness is an artifact computed by an untrusted algorithm. We formally define a witness in Sect. 2.3.

For our formalization, we assume that an input assigns values to variables, and analogously, for an output and potential witness. A variable gets assigned exactly one value for a sub-input. An input is composed of all sub-inputs, and thus, in contrast, the same variable may get assigned multiple values. That is why, we distinguish two types of assignments for our formalization. For sets A and B , a function $f : A \rightarrow B$ is an *assignment* of A in B . A relation $r \subseteq A \times B$ is a *weak assignment* of A in B . We denote the set of all assignments of A in B as $[A]$ and the set of all weak assignments of A in B as $\llbracket A \rrbracket$ (assuming B from the context).

Let I , O and W be finite sets of variables for the input, the output and the potential witness, respectively. For readability, we use different sets even though they do not have to be disjoint. We assume subsets $I_v \subseteq I$, $O_v \subseteq O$ and $W_v \subseteq W$ of variables for each component $v \in V$ such that $I = \cup_{v \in V} I_v$, $O = \cup_{v \in V} O_v$ and $W = \cup_{v \in V} W_v$. Let Val be a set of values. For each $v \in V$, let the sets of assignments $[I_v]$, $[O_v]$ and $[W_v]$ in Val be the sets of sub-inputs, sub-outputs, and sub-witnesses. Let the sets of weak assignments $\llbracket I \rrbracket$, $\llbracket O \rrbracket$ and $\llbracket W \rrbracket$ in Val be the sets of inputs, outputs and potential witnesses. The following holds for an input: if we have a sub-input $i_v \in [I_v]$ for each $v \in V$, then the weak assignment

$i = \cup_{v \in V} i_v$ is the according input. The same holds each for an output and a potential witness.

In the sequel, we fix

- the graph G as the network topology,
- the set Val as a domain,
- the sets of weak assignments $\llbracket I \rrbracket$, $\llbracket O \rrbracket$ and $\llbracket W \rrbracket$ in Val as inputs, outputs and potential witnesses,
- and the sets $[I_v]$, $[O_v]$ and $[W_v]$ in Val for each $v \in V$ as sub-inputs, sub-outputs, and sub-witnesses of v .

Moreover, we assume the minimal sub-input of a component is its own ID and the IDs of its neighbors in the network graph. Hence, the minimal input is the network itself.

2.2 Example: Witness for a Bipartite Network

As an example, consider distributed bipartite testing [5] where the components decide together whether the underlying network graph is bipartite (i.e. its vertices can be divided into two partitions such that each edge has a vertex in each partition). The input is the network itself presented by the sub-input of each component: the component's ID and the IDs of its neighbors in the network. In the case of a bipartite network, the sub-output of each component is 'true'. While in the case of a non-bipartite network, some components have the sub-output 'false' and the other components 'true'. In either case, the output is composed of those sub-outputs.

We consider a certifying variant of distributed bipartite testing. It follows from the definition of bipartiteness that a bipartition of the network's components is a witness for a network being bipartite. The witness is distributed in the way that each component has a bipartition of its neighborhood as a sub-witness. For the more sophisticated witness of a non-bipartite network, see [28].

For a better understanding of the formalization, consider the concrete network shown in Fig. 1 where e.g. the sub-input $i_3 \in [I_3]$ of component 3 assigns the value $\{6\} \in P(V)$ to the variable $\text{nbrs}_3 \in I_3$. In the remainder of this Section, we refer to this example to illustrate concepts.

2.3 Witness Predicate

For the problem to be solve by a terminating distributed algorithm, we assume a specification given as a precondition $\phi \subseteq \llbracket I \rrbracket$ and a postcondition $\psi \subseteq \llbracket I \rrbracket \times \llbracket O \rrbracket$. In the following, we fix the specification over input-output pairs as

$$\forall i \in \llbracket I \rrbracket, o \in \llbracket O \rrbracket : \psi(i, o) \vee \neg \phi(i)$$

We define a witness predicate over inputs, outputs and potential witnesses for the ϕ - ψ specification, and define the notion of a witness:

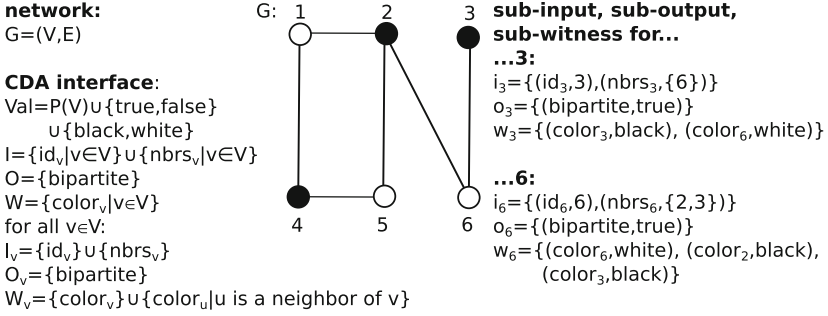


Fig. 1. Example of a bipartite network with the CDA interface and the sub-input, sub-output and sub-witness of components 3 and 6. $P(V)$ denotes the power set of V .

Definition 1 (witness predicate, witness, complete).

(i) A predicate $\Gamma \subseteq \llbracket I \rrbracket \times \llbracket O \rrbracket \times \llbracket W \rrbracket$ with the witness property

$$\forall i \in \llbracket I \rrbracket, o \in \llbracket O \rrbracket, w \in \llbracket W \rrbracket : \Gamma(i, o, w) \longrightarrow (\psi(i, o) \vee \neg\phi(i))$$

is a witness predicate for a ϕ - ψ specification.

(ii) If $(i, o, w) \in \Gamma$, w is a witness for the correctness of (i, o) .

(iii) Γ is a complete witness predicate if additionally holds

$$\forall i \in \llbracket I \rrbracket, o \in \llbracket O \rrbracket, w \in \llbracket W \rrbracket : \Gamma(i, o, w) \longleftarrow (\psi(i, o) \vee \neg\phi(i))$$

Note that an algorithm computes a *potential* witness w since it may be that $(i, o, w) \notin \Gamma$. However, if clear from context, we simply say witness from now on.

The witness predicate of the bipartite example states that the witness is a bipartition in the network. Its witness property follows by the definition of bipartiteness. Since the witness predicate holds with a biimplication, it is complete.

2.4 Consistency of a Distributed Witness

In the bipartite example, a sub-witness contains the colors of the neighbours – a bipartition of the neighborhood. Note that the sub-witnesses of neighbors have some common variables. In the example shown in Fig. 1, the components 3 and 6 have the variable $color_3$ in common. Consequently, in order to form a bipartition in the network, the common variables have to be consistent in their assignment.

In the general case, all sub-witnesses have to be consistent with each other in order to form a proper argument for the correctness of an input-output pair.

Definition 2 (consistent). Let $w \in \llbracket W \rrbracket$ be a witness.

(i) For $u, v \in V$: sub-witnesses $w_u \subseteq w$ and $w_v \subseteq w$ are consistent if and only if for all $a \in W_u \cap W_v$ holds $w_u(a) = w_v(a)$.

(ii) w is consistent if and only if $w \in \llbracket W \rrbracket$.

In the example of bipartite testing (Fig. 1), the sub-witnesses of components 3 and 6 have common variables: $W_3 \cap W_6 = \{color_3, color_6\}$. Since $w_3(color_3) = black = w_6(color_3)$ and $w_3(color_6) = white = w_6(color_6)$, the sub-witnesses w_3 and w_6 are consistent.

A witness is trivially consistent if for all $u, v \in V$ pairwise holds $W_u \cap W_v = \emptyset$. However, having a trivially consistent witness is often only possible by having a trivial distribution of the witness, since the witness is basically centralized, i.e. $W_v = W$ for one $v \in V$ and holds $W_u = \emptyset$ for all other $u \in V$. For instance, in the bipartite example, one component $v \in V$ has to have the whole bipartition of the network and network topology as a sub-witness then. Assume there is one other component u that has a part of the bipartition and the topology as its sub-witness. Then the two bipartitions presented in w_v and w_u have to be related to each other. Otherwise, the two bipartitions together may not form a bipartition. Hence, $W_v \cap W_u \neq \emptyset$ – a contradiction to the witness being trivially consistent. As a consequence, there are usually some components $u, v \in V$ with common variables in their sub-witnesses, i.e. $W_u \cap W_v \neq \emptyset$.

Lemma 1. *A witness is consistent if and only if all of its sub-witnesses are pairwise consistent.*

Proof. Let $w \in [W]$ be a consistent witness. Then for all $a \in W$ there is a unique value $w(a)$. Thus, for all $u, v \in V$ with $w_u, w_v \in w$ holds $w_u(a) = w(a) = w_v(a)$ if $a \in W_u \cap W_v$. Consequently, all sub-witnesses are pairwise consistent.

For the other direction, assume all sub-witnesses of $w \in [W]$ are pairwise consistent. For all $a \in W$, there is at least one component, w.l.o.g. $v \in V$, with $a \in W_v$ since $W = \cup_{v \in V} W_v$. For every component $u \in V$ with $a \in W_u$ holds $w_u(a) = w_v(a)$. Hence, $w \in [W]$ is consistent.

The need for consistency arises because the witness is *distributed*. Hence, certifying *sequential* algorithms do not have to deal with consistency (c.f. [19]). As a consequence, checking becomes more challenging for certifying *distributed* algorithms. To avoid checking consistency of all sub-witnesses pairwise, we restrict ourselves to a connected witness. We define a connected witness over all a -components:

Definition 3 (a -component). *If $a \in W_v$ for a component $v \in V$, then v is an a -component.*

Definition 4 (connected). *A witness $w \in [W]$ is connected if for all $a \in W$, the sub-graph induced by the a -components is connected.*

In the example shown in Fig. 1, the witness is connected. For instance, the components 2, 3 and 6 are the $color_6$ -components and they induce a connected sub-graph.

As an example for a witness that is not connected, assume a bipartite network where components belonging to the same partition solve one task together. Moreover, assume a part of this task is agreeing on some choice with one consent (i.e. a consensus problem [17]). In order to verify that all components of one

partition agree on their choice, the sub-witness of a component consists of its own choice and of the choices of the components in 1-hop-distance – components that share a neighbor are in 1-hop-distance. For example in Fig. 1, component 3 is in 1-hop-distance of component 2. The components in 1-hop-distance always belong to the same partition. The witness predicate is satisfied if each component agrees on its choice with the components in a 1-hop-distance. The witness is not connected since only components of the same partition share variables in their sub-witnesses, and therefore do not induce a connected subgraph.

Lemma 2. *Let $\Gamma \subseteq \llbracket I \rrbracket \times \llbracket O \rrbracket \times \llbracket W \rrbracket$ be a predicate. For every triple $(i, o, w) \in \Gamma$ where w is not connected, there is a triple $(i, o, w') \in \Gamma$ where w' is connected.*

Proof. Since $w \in \llbracket W \rrbracket$ is not connected, there are components $u, v \in V$ with $a \in W_u \cap W_v$ such that there is no path $p = (u, x_1, x_2, \dots, x_m, v)$ between u and v with all components x_l on the path having $a \in W_{x_l}$ for $l = 1, 2, \dots, m$.

We construct a connected witness w' from w . We add for each such outlined pair of components u, v on one path between u and v the missing variables $a \in W_u \cap W_v$. W.l.o.g. let this path be $p = (u, x_1, x_2, \dots, x_m, v)$. For each component x_l for $l = 1, 2, \dots, m$ is $W'_{x_l} := W_{x_l} \cup \{a\}$. It follows that w' is connected.

To ensure $(i, o, w') \in \Gamma$, we construct the sub-witnesses w'_{x_l} by adding the assignments of u (or analogously v): $w'_{x_l} := w_{x_l} \cup \{(a, w_u(a)) \mid a \in W'_{x_l} \setminus W_{x_l}\}$ for all $l = 1, 2, \dots, m$. Since w and w' are the union of the sub-witnesses, it holds $w = w'$, and therefore $(i, o, w') \in \Gamma$.

For a connected witness, it is sufficient to check the consistency in each neighborhood.

Definition 5 (consistent neighborhood). *Let $w \in \llbracket W \rrbracket$ be a witness. $v \in V$ has a consistent neighborhood if and only if for all neighbors u of v holds the sub-witnesses $w_v \subseteq w$ and $w_u \subseteq w$ are consistent.*

Theorem 1. *Let $w \in \llbracket W \rrbracket$ be a connected witness. w is consistent if and only if the neighborhood is consistent for all $v \in V$.*

Proof. If w is consistent, then it follows from Lemma 1 that all sub-witnesses of w are pairwise consistent. Thus, for each $v \in V$ the neighborhood is consistent.

For the other direction, let $u, v \in V$ with $a \in W_u \cap W_v$. From the definition of a connected witness follows, there exists a path between the a -components u and v over a -components. Since on this path all neighboring components are consistent, it follows by transitivity that u and v are consistent. Thus, the witness w is consistent.

For some CDAs, a sub-witness of a component v holds variables of the sub-output of a component u , c.f. [28–30]. Revisit the example where the components of one partition in a bipartite network solve a consensus problem. The sub-output of a component is its own choice. Part of the sub-witness of a component is the choices of the components in 1-hop-distance. Hence, the sub-witness of a component consists partly of sub-outputs of other components. For the shared variables,

the sub-outputs and sub-witnesses have to be consistent in their assignments. Since we do not want to check the consistency between sub-witnesses and sub-outputs or sub-inputs, we define a complete witness:

Definition 6 (complete). *A witness $w \in \llbracket W \rrbracket$ is complete if for all $u, v \in V$ and all $a \in W_u$ holds if $a \in I_v \cup O_v$, then $a \in W_v$.*

Note that if for all $v \in V$ holds $i_v \subseteq w_v$ and $o_v \subseteq w_v$, then the witness is complete.

2.5 Distributable Witness Predicate

In Sect. 2.7, we present a distributed checker that decides the witness predicate. However, the Definition 1 of the witness predicate is defined over the input, output and potential witness of a CDA and does not take into account sub-inputs, sub-outputs and sub-witnesses of the components. For distributed checking of the witness predicate, we define predicates that are decided for each component over the sub-input, sub-output and sub-witness, and are then combined to decide the witness predicate (c.f. [28]). A witness predicate is distributable in a network if some predicates hold for all components while others hold for at least one:

Definition 7 (distributable, completely).

(i) *Let $i \in \llbracket I \rrbracket$ be an input and its sub-inputs $i_v \in [I]$ for $v \in V$ such that $i = \cup_{v \in V} i_v$, let $o \in \llbracket O \rrbracket$ be an output and its sub-outputs $o_v \in [O]$ for $v \in V$ such that $o = \cup_{v \in V} o_v$, and let $w \in \llbracket W \rrbracket$ be a potential witness and its sub-witnesses $w_v \in [W]$ for $v \in V$ such that $w = \cup_{v \in V} w_v$. A predicate $\Gamma \subseteq \llbracket I \rrbracket \times \llbracket O \rrbracket \times \llbracket W \rrbracket$ is distributable if one of the following holds:*

1. *Γ is universally distributable with a predicate $\gamma \subseteq [I] \times [O] \times [W]$ if:
 $(\forall i_v \in [I_v], o_v \in [O_v], w_v \in [W_v] : \gamma(i_v, o_v, w_v)) \longrightarrow \Gamma(i, o, w)$.*
2. *Γ is existentially distributable with a predicate $\gamma \subseteq [I] \times [O] \times [W]$ if:
 $(\exists i_v \in [I_v], o_v \in [O_v], w_v \in [W_v] : \gamma(i_v, o_v, w_v)) \longrightarrow \Gamma(i, o, w)$.*
3. *There exist distributable predicates Γ_1, Γ_2 such that
 $(\Gamma_1(i, o, w) \wedge \Gamma_2(i, o, w)) \longrightarrow \Gamma(i, o, w)$.*
4. *There exist distributable predicates Γ_1, Γ_2 such that
 $(\Gamma_1(i, o, w) \vee \Gamma_2(i, o, w)) \longrightarrow \Gamma(i, o, w)$.*

(ii) *If the implications of 1-4 are also bimplications, then Γ is completely distributable.*

The predicates Γ_1 and Γ_2 “divide” the witness predicate in universally or existentially distributable predicates that are linked together by a conjunction or disjunction. We call the predicates Γ_1 and Γ_2 the *distribution-predicates* of Γ , and a predicate γ a *sub-predicate* of a universally or existentially distributable predicate.

Revisit the example of bipartite testing (Sect. 2.2), the witness predicate holds if the witness is a bipartition of the network. This witness predicate is universally distributable with a distribution-predicate that is satisfied if there is

a bipartition of the neighborhood for all components, and a sub-predicate stating that the sub-witness of component is a bipartition of the neighborhood. For an example of a not simply universally distributable witness predicate, see [28].

Note that not every predicate is distributable since we allow only conjunction and disjunction of distributable predicates (see rules 3 and 4). As a consequence, we cannot form a nesting of quantifiers for instance. However, the chosen restrictions enable us to decide the sub-predicates γ for each component independently, and to evaluate distribution-predicates in the whole network by using a spanning tree (c.f. Sect. 2.7). A more complex structure than a spanning tree would be needed to evaluate nested quantification in the network.

2.6 A Class of Certifying Distributed Algorithms

We define a class of certifying distributed algorithm that terminate and verify their distributed input-output pair at runtime by a distributed witness such that the distributable witness predicate is decided by a distributed checker:

Definition 8 (Certifying Distributed Algorithm). *A certifying distributed algorithm solving a problem specified by a ϕ - ψ specification computes for each input $i \in \llbracket I \rrbracket$ an output $o \in \llbracket O \rrbracket$, and a witness $w \in \llbracket W \rrbracket$ in the way that each component $v \in V$ computes for a sub-input $i_v \in [I]$, a sub-output $o_v \in [O]$ and a sub-witness $w_v \in [W]$ such that $i = \cup_{v \in V} i_v$, $o = \cup_{v \in V} o_v$ and $w = \cup_{v \in V} w_v$. Let $\Gamma \subseteq \llbracket I \rrbracket \times \llbracket O \rrbracket \times \llbracket W \rrbracket$ be a complete witness predicate for a ϕ - ψ specification. The following holds:*

- (i) $(i, o, w) \in \Gamma$,
- (ii) Γ is completely distributable,
- (iii) w is consistent,
- (iv) w is complete with $i_v \subseteq w_v$ and $o_v \subseteq w_v$ for all $v \in V$, and
- (v) w is connected.

From (i) follows the correctness of the input-output pair (i, o) . With (ii), we enable distributed checking of Γ . Usually, there are some components $u, v \in V$ with common variables in their sub-witnesses, i.e. $W_u \cap W_v \neq \emptyset$. Hence, the distributed witness has to be consistent as stated in (iii). By having a complete and connected witness as stated in (iv) and (v), we enable distributed checking of the consistency of the witness. Note that a connected witness is no restriction on the kind of possible correctness arguments following from Lemma 2.

Remark 1. For every distributed algorithm solving a problem specified by ϕ and ψ , there is a certifying variant belonging to the outlined class. A terminating distributed algorithm can always compute a witness for a correct input-output pair, e.g. the history of computation and communication for each component. The witness predicate then is satisfied if the computation and communication is in accordance with the algorithm.

However, proving the witness property then becomes complete verification of the distributed algorithm. Hence, a challenge is to find a “good” witness (c.f. [19] for certifying *sequential* algorithms). Finding a witness is a creative task just like developing an algorithm. However, design patterns such as using characterizing theorems or a spanning tree help.

There are two perspectives on a CDA: the one of the developer and the one of the user. The developer proves the correctness of his/her algorithm. By the definition of a CDA, the developer has for instance to prove that the algorithm computes a witness *for all* input-output pairs. For the user, however, it is enough to be convinced that his/her *particular* input-output pair is correct. To this end, the user has to understand the witness property of the witness predicate and to understand that the witness predicate is distributable. The user does not have to understand that the witness predicate is complete or that it is completely distributable. If the witness predicate is satisfied, the particular input-output pair is correct; if not, the output or the witness is not correct. Consequently, using a CDA comes at the expense of incomplete correctness.

Since for a satisfied witness predicate, the user still has to trust in the witness property, we discuss machine-checked proofs for a reduced trust base in Sect. 3.

2.7 Distributed Checker of a Distributed Witness

Let Γ be a distributable witness predicate with distribution-predicates $\Gamma_1, \Gamma_2, \dots, \Gamma_k$ and according sub-predicates $\gamma_j, j = 1, 2, \dots, k$. For distributed checking of Γ , each component has a *sub-checker* that checks the completeness of its sub-witness, the consistency of the sub-witnesses in the neighborhood, decides the sub-predicates for its component, and plays its part in checking the connectivity of the witness, and in evaluating the witness predicate. We assume a sub-checker gets a trusted copy of the sub-input (c.f. [19]). After termination is detected (e.g. as in [25]), a sub-checker receives the sub-output and sub-witness of its component, and starts checking. We assume a spanning tree as a communication structure in the network. This spanning tree is either reused or computed as discussed in Sect. 3.

Completeness. For each $v \in V$, let the predicate $comp_v$ denote whether w_v is complete: $i_v \subseteq w_v$ and $o_v \subseteq w_v$. The sub-checker of v decides $comp_v(i_v, o_v, w_v)$.

Connectivity. For each variable $a \in W$ in each connected subgraph of a -components, the components select the a -component with the smallest ID as a leader: First, each component v suggests itself as a leader for all its variables $a \in W_v$ to its neighbors. If a component receives a message containing a suggestion of a smaller leader for one of its variables, it updates the leader and forwards the message to all neighbors. After detection of termination, each component v holds a list associating the according leader ID with each variable: $((a_1, v_1), (a_2, v_2), \dots, (a_m, v_m))$ with $a_j \in W_v, v_j \in V$ and $j = 1, 2, \dots, m$. Note that a component v does not forward a message if it receives a suggestion for a leader of a variable $a \notin W_v$. Thus, if there are two different leaders for the same variable a in the network, then the subgraph of a -components is unconnected

and thereby the witness is not connected. Deciding whether there are multiple leaders for one for one variable can be done by using a spanning tree. Since we use a spanning tree as well for deciding the witness predicate, we describe this step as part of the evaluation.

Consistency. For each $v \in V$, let the predicate $cons_v \subset [W_v] \times [W_{u1}] \times [W_{u2}] \times \dots \times [W_{ul}]$ denote whether the neighborhood of v is consistent with neighbors $u1, u2, \dots, ul \in V$. We assume the sub-checkers of neighbors can communicate with each other. It follows from Theorem 1 that the consistency of a connected witness can be decided by a distributed algorithm where a component only once exchanges messages with its neighbors. Each sub-checker sends the sub-witness of its component to the neighboring sub-checkers. Subsequently, a sub-checker of each component v compares the sub-witness w_v with each of the received sub-witnesses: If for all $a \in W_v \cap W_{ui}$, $w_v(a) = w_{ui}(a)$, then $cons_v(w_v, w_{u1}, w_{u2}, \dots, w_{ul})$ holds.

Sub-Predicates. Each sub-checker of a component $v \in V$ decides each sub-predicate $\gamma_1, \gamma_2, \dots, \gamma_k$ for the triple (i_v, o_v, w_v) . Finally, the sub-checker holds a k -tuple containing the according evaluated sub-predicates.

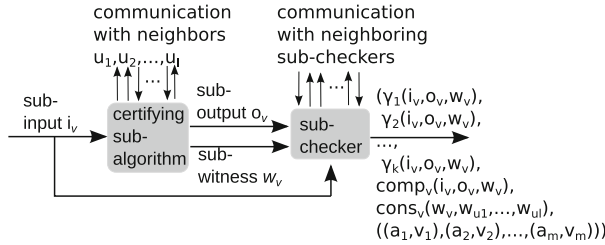


Fig. 2. A certifying sub-algorithm of $v \in V$ and its sub-checker.

Evaluation. Figure 2 shows a component with its sub-checker: Each sub-checker of a component v with neighbors $u1, u2, \dots, ul \in V$ holds a $k + 3$ -tuple consisting of k evaluated predicates, the evaluated predicates $comp_v$ and $cons_v$, and the list of associated leaders for each $a \in w_v$. To evaluate the witness predicate, the sub-checkers combine their tuples by using the rooted spanning tree: Starting by the leaves, each sub-checker gets the tuple of each child and combines it with its own tuple: if the j -th sub-predicate is universally distributable, then the j -th position of both tuples is combined by logical conjunction; otherwise the j -th sub-predicate is existentially distributable and logical disjunction is used instead. Let the predicate $Comp$ denote whether each sub-witness is complete and the predicate $Cons$ denote whether a witness w is consistent in the network; hence, both predicates are treated as universally distributable. For the connectivity, each component compares the chosen leaders of itself and its children. If a variable has multiple leaders, the component sends ‘false’ to its

parent otherwise a list with the so far chosen leaders. If a component receives false from a child, it just sends ‘false’ to its parent. Finally, the root creates the tuple $(\Gamma_1(i, o, w), \Gamma_2(i, o, w), \dots, \Gamma_k(i, o, w), \text{Comp}(w), \text{Cons}(w), \text{Con}(w))$ where the predicate Con is fulfilled if there are no multiple leaders for a variable – hence the witness is connected. The root evaluates the witness predicate by combining the distribution-predicates accordingly. The evaluation terminates when root receives a message from all its children; if the witness is complete, connected and consistent, and the witness predicate satisfied, the root accepts. All sub-checkers together build a distributed checker of Γ . From the definition of a CDA and the outlined distributed checker follows: if the distributed checker accepts on a triple (i, o, w) , then $(i, o) \in \psi$ or $(i) \notin \phi$.

3 Framework: Formal Instance Verification

We present a method for *formal instance verification* for CDAs (c.f. [29]). While formal verification establishes the correctness *for every* input-output pair at *design-time*, formal *instance* verification establishes the correctness *for a particular* input-output pair at *runtime*. In analogy to formal verification, formal instance verification requires a *machine-checked* proof. Hence, we have *formal instance correctness* for a particular input-output pair if there is a *machine-checked* proof for the correctness of this pair. While formal verification is often too costly, formal instance verification is often feasible but at the expense of not being complete.

To achieve formal instance correctness, we combine CDAs with theorem proving and program verification. We give an overview of the proof obligations to solve (Sect. 3.1). We implement the method in a framework for the proof assistant COQ (Sect. 3.2).

3.1 Proof Obligations for Formal Instance Verification

Using a CDA comes with a trust base: for example we have to trust that the witness predicate has the witness property or that the distributed checker algorithm is correct. According proofs have to be provided by the developer of the CDA but usually only exist on paper. Even if a distributed checker algorithm is correct on paper, the implemented distributed checker program could still be flawed. Assume a CDA with a witness predicate Γ , we have to solve the following proof obligations (PO) to obtain formal instance correctness:

PO I The implemented termination detection is correct.

PO II Witness predicate Γ has the following properties:

- (i) Γ has the witness property (c.f. Sect. 2.3)
- (ii) Γ is distributable (c.f. Sect. 2.5).

PO III The Theorem 1 for distributed checking of consistency (c.f. Sect. 2.3).

PO IV The implemented distributed checker is correct (c.f. Sect. 2.7):

- (i) Each sub-checker checks if its sub-witness is complete.

- (ii) Each sub-checker takes part in checking if the witness is connected.
- (iii) Each sub-checker checks the consistency sub-witnesses in the neighborhood.
- (iv) Each sub-checker decides the sub-predicates for its component.
- (v) Each sub-checker takes part in evaluation of Γ .

By solving these proof obligations, it follows: If the distributed checker accepts on an input, output and witness, we have a machine-checked proof that the particular input-output pair is correct. Note that the computation of the output is not mentioned in the proof obligations; the CDA is treated as a black box.

According to the concept of certifying algorithms the verification of the checker should be easier than verifying the actual algorithm. We note that for our class of CDA the checker has to perform five tasks making it seemingly complex. Note that, except for PO IV(iv), each task only needs to be verified once for the outlined class of CDA. As a consequence, the verification effort for each certifying algorithm is the same in the distributed setting as in the sequential setting.

3.2 Overview of the Framework

We use the proof assistant COQ [14] for theorem proving and program verification. COQ provides a higher-order logic, a programming language, and some proof automations. Even though COQ's programming language is not turing-complete (since every program halts), COQ implements a mechanism to extract programs to functional programming languages like OCAML. To model a network in COQ, we use the graph library GRAPH BASICS [8] for the topology, and the framework VERDI [31] for the communication. By using VERDI, we extract a distributed checker that can be deployed on a real network.

The framework is illustrated in Fig. 3. The network model and the CDA model are fundamental for all proof obligations. The network model consists of a formalization of the network's topology and communication. The CDA model consists of the CDA Interface – a formalization of the sub-input/output/witness and witness-predicate of a particular CDA – and a verified termination detection algorithm. We use theorem proving to show for the witness predicate Γ that it has the witness property and that it is distributable (PO II) as well as for the proof of Theorem 1 (PO III). We use program verification for the termination detection algorithm (PO I) as well as for the distributed checker (PO IV). Some proof obligations have to be proven for each CDA (indicated by an arrow), others have to be proven only once for the outlined class of CDAs. In this paper, we focus on the latter ones. Note that computation of a spanning tree is an implicit part of termination detection (PO I) and evaluation PO IV(v). Hence, it makes sense to verify the computation once and then to reuse the spanning tree. Verified COQ programs can be extracted to verified OCAML programs.

We formalized the network model and CDA interface, and solved the proof obligations that deal with the consistency of the witness (PO III and PO IV(iii)). We formalized the notion of consistency and solved PO III (proof of Theorem 1)

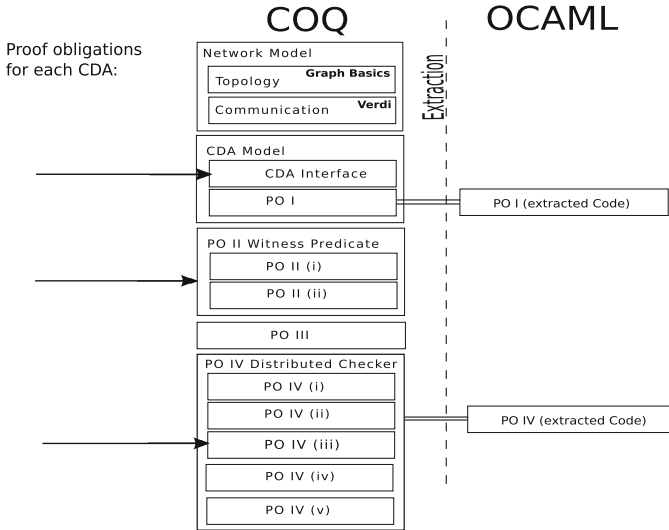


Fig. 3. A Coq framework for formal instance verification using CDAs.

in Coq. The formalization follows the definitions and the proof in Sect. 2.3 tightly. We forgo giving details in this paper. In the remainder of this section, we explain the network model and CDA interface (Sect. 3.3). We discuss the verification of the distributed consistency check (Sect. 3.4) and describe its extraction such that it runs on a real network (Sect. 3.5). PO IV(i), PO IV(ii), PO IV(v) follow the same approach as the distributed consistency check and are work-in-progress.

3.3 Network Model and CDA Interface

Topology. Since GRAPHBASICS offers a connected graph, the representation of a network is straightforward. We assume that a component and its checker are two *logical components* which are co-located on one *physical component*. A vertex of the connected graph **Component** represents the physical component.

Communication. We model the communication between a component and its sub-checker, and between sub-checkers. To implement the communication of the distributed checker, we specify the following definitions given by VERDI: The type of a sub-checker (**Name**), the set of sub-checkers (**Nodes**), the state each sub-checker maintains (**Data**) and a function to initialize this state (**initData**). VERDI distinguishes between internal (**Input** and **Output**) and external messages (**Msg**): While internal messages are exchanged between logical components running on the same physical component, external messages are exchanged across the network. We use internal messages for the communication between a component and its sub-checker, and external messages for the communication


```

1 Inductive Name := Checker : Component -> Name.
2 Definition Nodes : list Name := (map Checker (CV_list v a g)).
3 Inductive Msg := Checkermesssage : Certificate -> Msg.
4 Inductive Input : Type :=
5   | Checkerknowledge -> Input
6   | Checkerinput -> Input .
7
8 Record Data := mkData{
9   myCertificate : Certificate;
10  variables: list A;
11  nbrslist : list Component;
12  consistent : bool ;
13  initialized : bool;
14  (...)
15  }.
16
17 Definition initData (n: Name) := mkData
18   [] [] (neighbors g (name_component n)) false false (...).
19
20 Definition InputHandler (me : Name) (c : Input) (state: Data) :=
21 match me,c with
22 | Checker x, Checkerknowledge => if (initialized == false) then
23   variables := Checkerknowledge.(variables)
24   (...)
25 | Checker x, Checkerinput => if (initialized == false) then
26   myCertificate := Checkerinput.(certificate)
27   sendToAllNeighbors (mycertificate)
28   (...)
29
30 Definition NetHandler (me : Name) (src: Name) (m : Msg) (state: Data) :=
31 match m with
32 | Checkermesssage certificate => if (initialized == true) then
33   nbrslist := remove src state.(nbrslist)
34   consistent := state.(consistent) &&
35   Consistency_Nbr (certificate)

```

Fig. 4. Outline of the implementation of the consistency check of a sub-checker.

between sub-checkers. For the behaviour of a sub-checker, we implement the functions `InputHandler` and `NetHandler`. The `InputHandler` runs if a sub-checker receives an internal message and the (`NetHandler`) runs if a sub-checker receives an external message. For our network model, we assume reliable communication.

Combining Verdi and Graph Basics. VERDI does not offer to specify the topology of a network. However, to reason about properties such as consistency in a neighborhood, we have to specify the underlying topology of a network. That is why, we combine VERDI with GRAPH BASICS. To this end, we instantiate the set of `Nodes` in the network with the vertices of the topology graph.

CDA Interface. We abstract from the actual computation of a CDA. However, as a sub-checker needs to process sub-input, sub-output and sub-witness, we have to formalize them. The CDA interface consists of a formalization of the sub-input, sub-witness and sub-output as well as the structure of the witness predicate (i.e. if the distribution-predicates of the witness predicate are universally or existentially distributable). The latter is used by a sub-checker to perform the distributed evaluation of the witness predicate.

Initialization of a Sub-Checker. A sub-checker needs knowledge about its neighborhood; we implement the `initData` function (Fig. 4 l. 19) such that each

sub-checker is initialized with the IDs of its neighbors. Furthermore a sub-checker is initialized with the CDA Interface. We divide the CDA interface into two parts: The first part is independent from the actual computation of the CDA and contains the minimal sub-input and structure of the witness predicate. To this end, we define the internal message `Checkerknowledge`. The second part contains additional sub-input, the sub-output and sub-witness. To this end, we define the internal message `Checkerinput`. We define the `InputHandler` of a sub-checker such that it initializes the sub-checker's state with the values obtained from `Checkerknowledge` and `Checkerinput` (Fig. 4 l. 20–28).

3.4 Checking Consistency in the Neighborhood

To check the consistency of a witness, each sub-checker checks the consistency in its neighborhood (Theorem 1). In our implementation the state of each sub-checker contains a list of its neighbors (`nbrslist`). We use `nbrslist` to keep track of the messages received from the neighbors. Additionally, the state contains the boolean `initialized` which indicates if the sub-checker is initialized as described in the previous section, and the boolean `consistent` which indicates if the sub-witness of the component is consistent with all sub-witnesses received so far (Fig. 4 l. 10–17). When a sub-checker receives a sub-witness from a neighbor, it removes the neighbor from its `nbrslist`. As a result, if `nbrslist` is empty, a sub-checker received a sub-witness from each of its neighboring sub-checkers. Subsequently, a sub-checker calls the function `Consistency_Nbr` which takes two sub-witnesses as an input and returns true if they are consistent. If `Consistency_Nbr` returns true, the checker sets `consistent` to true. After being set to false once, the value of `consistent` cannot become true again. If the consistency check fails for at least one neighborhood, the witness is inconsistent.

Verification of the Consistency Check. For the verification of the consistency check, we show that if the consistency check succeeds, the neighborhood of each sub-checker is consistent. After initialization, if a sub-checker s received and processed a sub-witness from each neighboring sub-checker and `consistent` is true, consistency in the neighborhood of s holds:

Theorem 2 (in Coq). $\forall s, \text{initialized}(s) \wedge \text{nbrslist}(s) = \text{empty} \wedge \text{consistent}(s) \longrightarrow \text{Neighborhood Consistency}(s)$

We prove this theorem in the following steps using Coq. First, we show that for all reachable network states that the following lemmas hold for each sub-checker s :

Lemma 3 (in Coq). *All components in `nbrslist(s)` are neighbors of s .*

Lemma 4 (in Coq). *From `initialized(s)` follows that, if `nbrslist(s)` is empty, a message was received from each neighbor of s .*

Lemma 5 (in Coq). *From `initialized(s)` follows that, if `consistent(s)` is true, the witness is consistent with each witness received so far.*

We prove the Lemmas 1–3 by *inductive state invariants* [31]. A property is an inductive state invariant if it holds in the initial state (defined by the `initState` function – Fig. 4 l. 19) and each state reachable by processing a message. Note that Lemma 2 and 3 rely on the value of `initialized` which has nothing to do with VERDI’s `initState` function but with the initialization of our network model described in the previous section. As a next step, we verify the function `Consistency_Nbr` by proving that it returns true for two sub-witnesses if and only if the sub-witnesses are consistent. Finally, we show that the Lemmas 1–3 and the correctness of the function `Consistency_Nbr` together imply the correctness of Theorem 2.

3.5 Extraction of a Distributed Checker

To run our distributed checker on a real network we extract it to OCAML and link it with the VERDI SHIM – a small library which e.g. provides network primitives. In order to extract our distributed checker we have to provide a specific topology and instantiate the types of the CDA interface accordingly.

The trusted computing base of a distributed checker consists of the following: COQ’s proof checker and extraction mechanism – both proven on paper, the OCAML compiler – widely used, VERDI’s SHIM and the underlying operating system.

4 Related Work

Literature offers numerous certifying algorithms [1–4, 6, 9–13, 15, 18–20, 22–24, 27]. A theory of certifying algorithms and further reading is given in [19]. A formal instance verification method is discussed in [26]. All this work is on *sequential* algorithms. To the best of our knowledge, there is little research on certifying *distributed* algorithms. A certifying variant of a routing algorithm was presented in [30], a discussion on how to distribute a witness predicate over a network in [28], and a method for formal instance verification in [29]. Moreover, CDAs share similarities to self-stabilizing algorithms [7], proof labeling schemes [16], and decentralized runtime verification [21].

In this paper, we built up on previous work [28, 29]. We integrated the idea of a consistent witness, and focused on distributed checking of consistency and the witness predicate in contrast to [28]. Moreover, we discussed proof obligations that have to be proven only once for the outlined class of CDAs while in [29] one particular case study is discussed. Moreover, we integrated VERDI for verification of a *distributed* program. As a consequence, the verified distributed checker runs on a real network in contrast to [29].

5 Discussion

We considered CDAs which verify an input-output pair at runtime. There are many open questions on how to apply the concept of certification to distributed

algorithms. We focused on the *distributed checking* of a *distributed witness*. In order to form a valid correctness argument a distributed witness has to be *consistent*. By a restriction to *connected* witnesses, consistency can be checked in the neighborhoods (Theorem 1). We presented a method for formal instance verification, and implemented this method in a framework for COQ. Moreover, we discussed a verified implementation of the consistency check as an example of a task of the distributed checker. We showed how to deploy the verified distributed checker on a real network.

In the discussed framework, some proof obligations require manual work for each CDA (Sect. 3). For the proof obligations PO II(i) and PO II(ii) we have to find a proof. Automatic theorem provers can help to partly automate this undecidable task. However, using different tools creates an overhead: We have to formalize a proof obligation for different tools, and to show that the different formalizations are equivalent. Moreover, the tools add up on the trust base (c.f. [26]). For the proof obligation PO IV(iv) we have to verify the correctness of the checkers task to decide the sub-predicates. By restricting to simple sub-predicates, i.e. sub-predicates that can be expressed as a propositional logic formula, we could use a verified program that gets a sub-predicate and generates a decision procedure correct by construction. By implementing and verifying such a program in COQ, we could easily integrate it to the presented framework.

We focused on terminating distributed algorithms. However, some distributed algorithms are intended to run continuously such as communication protocols. On a synchronous network, each round could additionally consist of a checking phase. By restricting to a universally distributable witness predicate, a sub-checker can raise an alarm if a sub-predicate does not hold. If not restricting to universally distributable witness predicates, the overhead of the evaluation of the witness predicate could be reduced by evaluating each k rounds. As a consequence, a bug would be discovered with a possible delay.

We focused on networks. However, for shared memory systems, the consistency of a distributed witness could be guaranteed by sharing the according variables between neighbors. The witness still has to be connected however. An alternative is to have sub-checkers that act like an interface of its component. That way, a sub-checker could check whether all messages sent are consistent with the internal state of its component, that its component does not corrupt messages when forwarding them, and that its component reads out a message properly. By that, the computed witness would be consistent. However, an overhead would be created during the computation.

References

1. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C.: Verification of certifying computations. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 67–82. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_7
2. Alkassar, E., Böhme, S., Mehlhorn, K., Rizkallah, C.: A framework for the verification of certifying computations. *J. Autom. Reason.* **52**(3), 241–273 (2014)

3. Arkoudas, K., Rinard, M.C.: Deductive runtime certification. *Electron. Notes Theor. Comput. Sci.* **113**, 45–63 (2005)
4. Bruce, D., Hoàng, C.T., Sawada, J.: A certifying algorithm for 3-colorability of P^5 -free graphs. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) *ISAAC 2009*. LNCS, vol. 5878, pp. 594–604. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-10631-6_61
5. Censor-Hillel, K., Fischer, E., Schwartzman, G., Vasudev, Y.: Fast distributed algorithms for testing graph properties. In: Gavoille, C., Ilcinkas, D. (eds.) *DISC 2016*. LNCS, vol. 9888, pp. 43–56. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53426-7_4
6. Corneil, D.G., Dalton, B., Habib, M.: LDFS-based certifying algorithm for the minimum path cover problem on cocomparability graphs. *SIAM J. Comput.* **42**(3), 792–807 (2013)
7. Dolev, S.: *Self-Stabilization*. MIT Press, Cambridge (2000)
8. Duprat, J.: A Coq toolkit for graph theory. *Rapport de recherche* **15** (2001)
9. Finkler, U., Mehlhorn, K.: Checking priority queues. In: *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1999*, pp. 901–902. Society for Industrial and Applied Mathematics, Philadelphia (1999)
10. Glesner, S.: Program checking with certificates: separating correctness-critical code. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) *FME 2003*. LNCS, vol. 2805, pp. 758–777. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_41
11. Heggernes, P., Kratsch, D.: Linear-time certifying recognition algorithms and forbidden induced subgraphs. *Nord. J. Comput.* **14**(1), 87–108 (2007)
12. Hell, P., Huang, J.: Certifying LexBFS recognition algorithms for proper interval graphs and proper interval bigraphs. *SIAM J. Disc. Math.* **18**, 554–570 (2004)
13. Hung, R.W., Chang, M.S.: An efficient certifying algorithm for the Hamiltonian cycle problem on circular-arc graphs. *Theoret. Comput. Sci.* **412**(39), 5351–5373 (2011)
14. INRIA: The Coq Proof Assistant. <http://coq.inria.fr/>
15. Kaplan, H., Nussbaum, Y.: Certifying algorithms for recognizing proper circular-arc graphs and unit circular-arc graphs. *Disc. Appl. Math.* **157**(15), 3216–3230 (2009)
16. Korman, A., Kutten, S., Peleg, D.: Proof labeling schemes. *Distrib. Comput.* **22**(4), 215–233 (2010)
17. Lynch, N.A.: *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco (1996)
18. McConnell, R.M.: A certifying algorithm for the consecutive-ones property. In: *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004*, pp. 768–777. Society for Industrial and Applied Mathematics, Philadelphia (2004)
19. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. *Comput. Sci. Rev.* **5**, 119–161 (2011)
20. Mehlhorn, K., Näher, S.: From algorithms to working programs: on the use of program checking in LEDA. In: Brim, L., Gruska, J., Zlatuška, J. (eds.) *MFCS 1998*. LNCS, vol. 1450, pp. 84–93. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0055759>
21. Mostafa, M., Bonakdarpour, B.: Decentralized runtime verification of LTL specifications in distributed systems. In: *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015*, pp. 494–503. IEEE Computer Society, Washington, DC (2015)

22. Necula, G.C., Lee, P.: The Design and implementation of a certifying compiler. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI 1998, pp. 333–344. ACM, New York (1998)
23. Nikolopoulos, S.D., Palios, L.: An $O(nm)$ -time certifying algorithm for recognizing HHD-free graphs. *Theor. Comput. Sci.* **452**, 117–131 (2012)
24. Noschinski, L., Rizkallah, C., Mehlhorn, K.: Verification of certifying computations through autocorres and simpl. In: Badger, J.M., Rozier, K.Y. (eds.) NFM 2014. LNCS, vol. 8430, pp. 46–61. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06200-6_4
25. Raynal, M.: Distributed Algorithms for Message-Passing Systems. Springer, Heidelberg (2013). <https://doi.org/10.1007/978-3-642-38123-2>
26. Rizkallah, C.: Verification of program computations. Ph.D. thesis (2015)
27. Schmidt, J.M.: Construction sequences and certifying 3-connectivity. *Algorithmica* **62**, 192–208 (2012)
28. Völlinger, K.: Verifying the output of a distributed algorithm using certification. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 424–430. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_29
29. Völlinger, K., Akili, S.: Verifying a class of certifying distributed programs. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 373–388. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_27
30. Völlinger, K., Reisig, W.: Certification of distributed algorithms solving problems with optimal substructure. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 190–195. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22969-0_14
31. Wilcox, J.R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M.D., Anderson, T.: Verdi: a framework for implementing and formally verifying distributed systems. In: ACM SIGPLAN Notices, vol. 50, pp. 357–368. ACM (2015)



Preserving Contract Satisfiability Under Non-monotonic Composition

Jonas Westman¹(✉) and Mattias Nyberg^{1,2}

¹ Royal Institute of Technology (KTH), Stockholm, Sweden
jowestm@kth.se

² Scania, Södertälje, Sweden

Abstract. A contracts theory embeds *non-monotonic composition* (with respect to implementation) if the fact that a composition of two components implements a specification \mathcal{S} does not generally follow from one of these components implementing \mathcal{S} . In contrast to monotonic composition, non-monotonic composition offers the additional expressiveness of specifying properties that *only* hold locally for a component since non-monotonic composition does not enforce all properties to be preserved when composing. Despite that this additional expressiveness is clearly needed, it implies that cases where monotony is indeed desired needs to be managed explicitly. The present paper elaborates on this topic by introducing a contracts theory embedding non-monotonic composition, and exploring conditions for ensuring monotonic composition in the context of this theory.

Keywords: Contracts · Non-monotonic · Composition · Satisfiability

1 Introduction

The notion of *contracts* was first introduced in [10] as a pair of pre- and post-conditions [8] to be used in formal specification of software (SW) *components*. More recent, general contracts theories [2, 6, 9, 11–13] consider a wider notion of a contract applicable for components in any domain (SW, hardware, physical, etc.). In such general theories, a contract $(\mathcal{A}, \mathcal{G})$ expresses a commitment of a component to guarantee the *implementation* of a specified property \mathcal{G} , given that the component is *composed* with an environment implementing a specified property \mathcal{A} .

More concretely, despite using different terminology and notation, such general contracts theories [2, 6, 9, 11–13] are all grounded in the following concepts:

- component C ;
- composition \times on pairs of components;
- specification \mathcal{S} ; and
- implementation relation \models on components and specifications.

Defined from these concepts, a contract is a pair $(\mathcal{A}, \mathcal{G})$ of specifications and is *satisfied* by a component C if $E \times C \models \mathcal{G}$ for each component $E \models \mathcal{A}$.

Another common characteristic of general contracts theories [2, 6, 9, 11–13] is that they embed *monotonic composition* (with respect to implementation), i.e.

$$C \models \mathcal{S} \Rightarrow \forall C'. C' \times C \models \mathcal{S}. \quad (1)$$

An example is the theory in [2] where components and specifications both are characterized by *assertions*, i.e., sets of value sequences, over the same (universal) set of variables. Composition \times and implementation \models are instantiated by \cap and \subseteq , respectively, which means that (1) translates to $C \subseteq \mathcal{S} \Rightarrow \forall C'. C' \cap C \subseteq \mathcal{S}$, which generally holds.

Despite previous general contracts theories [2, 6, 9, 11–13] embedding monotonic composition, there is at least one strong reason why a general contracts theory by itself should not enforce (1), i.e., why a contracts theory should embed *non-monotonic composition* instead.

This reason is that theories embedding monotonic composition is inherently limited to specification of properties that are preserved through composition. Formulated differently, a theory with monotonic composition does not support specifying properties that *only* hold locally for a component since all properties are preserved when composing. This means that it is not possible to express, e.g., that a component *shall not* constrain a variable x to any particular value, but that the composition of this component and another *shall* constrain x . In an engineering context, this could correspond to a specification expressing that “a subsystem shall not send out a particular signal” – a property that will not be preserved when composed with a component that sends out the signal.

Thus, in contrast to embedding monotonic composition, embedding non-monotonic composition in a contracts theory offers the additional expressiveness of specifying purely local properties. Despite that this additional expressiveness is clearly needed, there are many cases where we do want composition to be monotonic with respect to implementation. Since a theory embedding non-monotonic composition does not ensure this by itself, in such a theory, it is necessary to understand the conditions for when composition is monotonic and when it is not.

More explicitly, in a theory embedding non-monotonic composition, consider that the guarantee of a contract $(\mathcal{A}, \mathcal{G})$ expresses a property intended to be *global*, i.e. always preserved through composition. If a component C satisfying $(\mathcal{A}, \mathcal{G})$ is first composed with a component $E \models \mathcal{A}$, it follows that $E \times C \models \mathcal{G}$; however, if $E \times C$ is later composed with another component C' , it does not generally follow that $C' \times E \times C \models \mathcal{G}$, and thus, this has to be ensured explicitly. More generally, independent of the order in which components are composed, it needs to be ensured that for each $E \models \mathcal{A}$ and C satisfying $(\mathcal{A}, \mathcal{G})$, it holds that $C' \times E \times C \models \mathcal{G}$, which is equivalent to

$$\forall C. (C \text{ satisfies } (\mathcal{A}, \mathcal{G}) \Rightarrow C' \times C \text{ satisfies } (\mathcal{A}, \mathcal{G})). \quad (2)$$

Since (2) is not embedded in a contracts theory with non-monotonic composition, ensuring (2) needs to be done explicitly. The present paper elaborates on

this topic by introducing a contracts theory embedding non-monotonic composition, and exploring conditions for ensuring (2) in the context of this theory.

The introduction of this contracts theory constitutes a first contribution. Similar to the previously exemplified theory [2], in the introduced theory, components are characterized by assertions and composition is set intersection; however, in contrast to [2], specifications are sets of assertions instead of just assertions, and implementation is instantiated by \in instead of \subseteq . The proposed theory embeds non-monotonic composition since (1) instantiates to $C \in \mathcal{S} \Rightarrow \forall C'. C' \cap C \in \mathcal{S}$, which clearly does not generally hold.

As a second contribution, conditions for ensuring (2) are presented. This includes a sufficient condition, which is that component C' satisfies contract $(\mathcal{G}|\mathcal{G})$. It is shown that this condition is in fact also necessary if, for each $G \in \mathcal{G}$, there exists a component that satisfies $(\mathcal{A}, \{G\})$. Furthermore, it is shown that monotonic composition with respect to composition can also be ensured by a condition based on having components C , C' , and each $E \in \mathcal{A}$ implement special types of specifications, called *interface port specifications*, which ensure that the set of variables constrained by components C' and $E \cap C$ are respectively disjoint, and thus, that the components are isolated from each other.

As previously mentioned, general contracts theories [2, 6, 9, 11–13] all embed monotonic composition, instead of non-monotonic composition as in the contracts theory presented in this paper. However, there are contracts meta theories [1, 3] that are not limited to monotonic composition. More specifically, these meta theories support instantiating theories embedding non-monotonic composition, but also contracts theories embedding monotonic composition. Thus, these meta theories are defined at a level agnostic to whether composition is monotonic or non-monotonic, which means that these meta theories naturally do not support reasoning about properties that are specific to contracts theories embedding non-monotonic composition. In contrast, the present paper studies the concept of non-monotonic composition in depth, presenting several theorems and propositions manifesting properties inherent to this concept.

The rest of the paper is organized as follows. Section 2 presents established concepts from previous contracts theories. Section 3 introduces, as a first contribution, a novel contracts theory where composition is non-monotonic with respect to implementation. Section 4 then presents, as a second contribution, conditions to ensure monotony of composition in this contracts theory. Section 5 summarizes the paper and draws conclusions.

2 Preliminaries

This section presents definitions already established in [13], which in turn draws on the contracts theory in [2].

Let $\Xi = \{x_1, \dots, x_N\}$ denote the set of variables considered, and we call this the *universal set* of variables. Similarly, let $T_{\geq 0} \subseteq \mathbb{R}$ denote the universal set of time points considered. $T_{\geq 0}$ can be defined to be a continuous, e.g. if $T_{\geq 0}$ is defined by $[0, 1000]$, or discrete, e.g. if $T_{\geq 0} = \{0.1 * n | n \in \{0, 1, 2, \dots, 10000\}\}$.

The time points in $T_{\geq 0}$ do not represent absolute time, but instead, time relative to the start of execution or observation of the system. Furthermore, $T_{\geq 0}$ contains 0 and no negative time points.

Definition 1 (Run, see [13]). A run $\omega = (x_1(t), \dots, x_N(t))$ is a vector valued function, with the elements being the variables in Ξ , and defined on a time subinterval $T^\omega \subseteq T_{\geq 0}$ starting at time 0, i.e. $0 \in T^\omega$. □

For example, a run can be a *trace* [5, 7, 14] or an *execution* as presented in [11]. Note that two different runs may be defined on two different time intervals.

Let Ω denote the set of all possible runs.

Definition 2 (Behavior, see [13]). A behavior B is a, possibly empty, set of runs. □

This notion corresponds to an *assertion* as presented in [2, 4].

Example 1. Consider a universal set of variables $\Xi = \{x, y\}$. Examples of runs are $\omega_1 = (x(t), y(t)) = (t, e^t)$ and $\omega_2 = (t, 2e^t)$ defined on an interval $T^\omega = [0, 10]$. These two runs can be combined to form four different behaviors $B_1 = \{\}$, $B_2 = \{(t, e^t)\}$, $B_3 = \{(t, 2e^t)\}$, and $B_4 = \{(t, e^t), (t, 2e^t)\}$. □

Example 2. Let $\Xi = \{x, y\}$ where x and y are Boolean variables and $T^\omega = \{0\}$ for all runs. Examples of behaviors are $B_1 = \{(0, 0)\}$, $B_2 = \{(0, 1)\}$, $B_3 = \{(0, 0), (0, 1)\}$, and $B_4 = \{(0, 0), (1, 1)\}$. □

Let $\omega[X]$ denote the subvector of ω having the variables $X \subseteq \Xi$. For example, if $\Xi = \{a, b, c, d\}$, then $\omega[\{b, d\}] = (b, d)$. If $X = \emptyset$, then let $\omega[X] = \emptyset$.

Definition 3 (Projection, see [13]). The projection of behavior B onto a set of variables $X \subseteq \Xi$, denoted $\text{proj}_X B$, is the set of runs:

$$\{\omega \in \Omega \mid \exists \nu \in B. \nu[X] = \omega[X]\}. \quad \square$$

Note that projection is called *extended projection* in [13] and is defined using slightly different notation. Note also that if $B \neq \emptyset$, then $\text{proj}_\emptyset B = \Omega$ and $\text{proj}_\Xi B = B$. Furthermore, for any variable set Y , it holds $\text{proj}_Y \emptyset = \emptyset$ and $\text{proj}_Y \Omega = \Omega$.

Example 3. For the behaviors defined in Example 2, it holds that $\text{proj}_{\{x\}} B_1 = \text{proj}_{\{x\}} B_2 = \{(0, 0), (0, 1)\}$ and $\text{proj}_{\{x\}} B_3 = B_3$, and $\text{proj}_{\{x\}} B_4 = \Omega$. □

Let $\{x\}^c$ denote the complement set of $\{x\}$, i.e. $\{x\}^c = \Xi \setminus \{x\}$.

Definition 4 (Behavior Constrains, see [13]). A behavior B constrains a variable x if

$$\text{proj}_{\{x\}^c} B \neq B. \quad \square$$

Let $X_{\mathbf{B}}$ denote the set of variables constrained by \mathbf{B} . Note that $X_{\Omega} = \emptyset$ and $X_{\emptyset} = \emptyset$.

Example 4. For the behaviors discussed in Examples 2 and 3, it holds $\{y\}^c = \{x\}$. Furthermore, $\text{proj}_{\{y\}^c} \mathbf{B}_1 = \text{proj}_{\{x\}} \mathbf{B}_1 \neq \mathbf{B}_1$. Therefore \mathbf{B}_1 constrains x . In fact, \mathbf{B}_1 also constrains y , and therefore $X_{\mathbf{B}_1} = \{x, y\}$.

Since $\text{proj}_{\{x\}} \mathbf{B}_3 = \mathbf{B}_3$, it holds that \mathbf{B}_3 does not constrain y . However, behavior \mathbf{B}_3 does constrain x since $\text{proj}_{\{y\}} \mathbf{B}_3 = \{(0, 0), (0, 1), (1, 0), (1, 1)\} \neq \mathbf{B}_3$. \square

3 Composition, Specifications, and Contracts

Based upon the definition of behavior in Sect. 2 and as the first contribution of the paper, this section presents the basis of a contracts theory in which specifications are represented as sets of behaviors. The achievement is a theory where composition is non-monotonous with respect to implementation of specifications.

3.1 Composition

We assume that each component is characterized by a behavior. When two components, characterized by behaviors \mathbf{B}_1 and \mathbf{B}_2 respectively, are put together, we assume that a new behavior, called *the composition of \mathbf{B}_1 and \mathbf{B}_2* , is formed and that it conforms to the following definition.

Definition 5 (Composition of Behaviors). *The composition of two behaviors \mathbf{B}_1 and \mathbf{B}_2 is their intersection $\mathbf{B}_1 \cap \mathbf{B}_2$.* \square

3.2 Specifications

A *specification* \mathcal{S} expresses an intended property of a component. Commonly such an intended property is called a ‘requirement’. If a behavior *implements* a specification, then this really means that the component characterized by the behavior has the intended property expressed by \mathcal{S} . These notions are now formalized in the following two definitions.

Definition 6 (Specification). *A specification \mathcal{S} is a, possibly empty, set of behaviors.* \square

Definition 7 (Behavior Implements Specification). *A behavior \mathbf{B} implements a specification \mathcal{S} if $\mathbf{B} \in \mathcal{S}$.* \square

Note that the empty set \emptyset can be a specification, and considering that \emptyset can also be a behavior, the set $\{\emptyset\}$ can be a specification.

Example 5. Let $\Xi = \{x, y\}$ and $T^\omega = \{0\}$. Then

$$\mathcal{S} = \{\{(0, 0), (0, 1)\}, \{(0, 1)\}, \{(0, 0), (1, 1)\}\}$$

is a possible specification. Behavior $\mathbf{B}_3 = \{(0, 0), (0, 1)\}$ implements \mathcal{S} and also $\mathbf{B}_4 = \{(0, 0), (1, 1)\}$ implements \mathcal{S} . However, the composition $\mathbf{B}_3 \cap \{(0, 0), (1, 1)\} = \{(0, 0)\}$ does not implement \mathcal{S} . \square

Example 5 highlights the fact that even though the two behaviors B_3 and B_3 individually implement the specification \mathcal{S} , their composition does not. That is, composition is non-monotonic with respect to implementation of specification.

3.3 Contracts

In alignment with the notion of *contract* and *satisfy* as introduced in Sect. 1, this section presents the instantiation resulting from the definitions of specification and implementation presented in Sect. 3.2.

A contract is a pair $(\mathcal{A}, \mathcal{G})$ of specifications expressing an intended property of a composition of two behaviors E and B where $E \in \mathcal{A}$.

Definition 8 (Contract). *A contract is a pair $(\mathcal{A}, \mathcal{G})$ of specifications, where:*

- (i) \mathcal{G} is non-empty and called *guarantee*; and
- (ii) \mathcal{A} is possibly empty and called *assumption*. □

Definition 9 (Behavior Satisfies Contract). *A behavior B satisfies a contract $(\mathcal{A}, \mathcal{G})$ if, for each E implementing \mathcal{A} , it holds that the composition $E \cap B$ implements \mathcal{G} . □*

If the instantiations of *specification* and *implements* are written out, we obtain that a behavior B satisfies a contract $(\mathcal{A}, \mathcal{G})$ if, and only if,

$$\forall A \in \mathcal{A}. A \cap B \in \mathcal{G} \tag{3}$$

Note that a contract $(\mathcal{A}, \mathcal{G})$ is trivially satisfied if $\mathcal{A} = \emptyset$.

3.4 Specification Is Agnostic to Set of Variables

As discussed above, a consequence of Definitions 6 and 7 of *specification* and *implements* is that composition is not inherently monotonic with respect to implementation. However, in many cases monotony is indeed desirable. One such case is when a specification represents an intended relation on a proper subset of the universal set Ξ . An example is when \mathcal{S}_1 expresses an intended property $x = 0$ and where $x \subset \Xi$. Consider a first component that itself implements \mathcal{S}_1 by setting $x = 0$ and not constraining any other variables. If this first component is composed with a second component that does not in any way constrain x , then we should expect that their composition also implements \mathcal{S}_1 . As a fundamental concept to support such reasoning, we introduce the notion of *agnostic specification*:

Definition 10 (Specification is Agnostic to Set of Variables). *A specification \mathcal{S} is agnostic to a set of variables X if*

$$\forall B \in \mathcal{S}. (\forall B'. \text{proj}_{X^c} B' = \text{proj}_{X^c} B \Rightarrow B' \in \mathcal{S}).$$

□

A specification is agnostic to a set of variables X if the fact whether or not a behavior implements the specification is independent on whether or not the behavior constrains variables in X .

Example 6. Let $\Xi = \{x, y\}$ where x and y are Boolean variables and $T^\omega = \{0\}$. Two examples of specifications that are agnostic to $\{y\}$ are:

- $\{\{(0, 0)\}, \{(0, 1)\}, \{(0, 0), (0, 1)\}\}$
- $\{\{(0, 0)\}, \{(0, 1)\}, \{(0, 0), (0, 1)\}, \{(1, 1)\}, \{(1, 0)\}, \{(1, 0), (1, 1)\}\}$.

Any proper subset of these behavior sets are *not* agnostic to $\{y\}$. Three such examples are:

- $\{\{(0, 0)\}\}$
- $\{\{(0, 0)\}, \{(0, 1)\}\}$
- $\{\{(0, 0), (0, 1), (1, 0), (1, 1)\}\}$. □

Now, to illustrate how the concept of agnostic specification can be used to ensure monotonic composition with respect to implementation of specification, consider again the previously introduced specification \mathcal{S}_1 . Assume that \mathcal{S}_1 , expressed by $x = 0$, is agnostic to $\{y\}$. Since the first component implements \mathcal{S}_1 , it holds $B_1 \in \mathcal{S}_1$. Also, since the first component does not constrain any other variables than x , it holds that B_1 does not constrain y . For any behavior B_2 constraining *only* y , it holds $\text{proj}_{\{x\}} B_2 = \Omega$ (see Proposition 2 in Appendix A). Next, we will use the following proposition with proof given in Appendix A:

Proposition 1. *Let B_1 and B_2 be two behaviors such that $X_{B_1} \cap X_{B_2} = \emptyset$. Then, for an arbitrary set of variables Y , it holds*

$$\text{proj}_Y(B_1 \cap B_2) = \text{proj}_Y B_1 \cap \text{proj}_Y B_2.$$

□

By using Proposition 1, we derive that

$$\text{proj}_{\{x\}}(B_1 \cap B_2) = \text{proj}_{\{x\}} B_1 \cap \text{proj}_{\{x\}} B_2 = \text{proj}_{\{x\}} B_1 \cap \Omega = \text{proj}_{\{x\}} B_1 \quad (4)$$

Specification \mathcal{S}_1 being agnostic to $\{y\}$ means according to Definition 10 that

$$\forall B'. \text{proj}_{\{x\}} B' = \text{proj}_{\{x\}} B_1 \Rightarrow B' \in \mathcal{S}_1 \quad (5)$$

Combining (4) and (5), implies that the composition $B' = B_1 \cap B_2 \in \mathcal{S}_1$.

In summary, the fact is that since \mathcal{S}_1 , expressing $x = 0$, is agnostic to y , the composition of any behavior in \mathcal{S}_1 with any other behavior that only constrains y will result in a behavior also in \mathcal{S}_1 . This fact will be generalized in Sect. 4.3 presenting a similar condition, but in terms of satisfying a contract instead of implementing a specification.

4 Preserving Contract Satisfiability Under Non-monotonic Composition

Section 3 presented a contracts theory where components are characterized by behaviors, composition is set intersection, specifications are behavior sets, and implementation is defined by \in . As previously mentioned in Sect. 1, the proposed theory embeds non-monotonic composition since (1) instantiates to $\mathbf{B} \in \mathcal{S} \Rightarrow \forall \mathbf{B}'. \mathbf{B}' \cap \mathbf{B} \in \mathcal{S}$, which clearly does not generally hold. As also stated in Sect. 1, this means that it follows that (2) also does not hold, i.e., it does not hold that

$$\forall \mathbf{B}. \mathbf{B} \text{ satisfies } (\mathcal{A}, \mathcal{G}) \Rightarrow \mathbf{B}' \cap \mathbf{B} \text{ satisfies } (\mathcal{A}, \mathcal{G}). \quad (6)$$

However, ensuring (6) is desirable for any use-case where \mathcal{G} expresses a property that is intended to be implemented by a composition of \mathbf{B}' and the composition $\mathbf{B} \cap \mathbf{E}$ of a component \mathbf{B} satisfying $(\mathcal{A}, \mathcal{G})$ and a component $\mathbf{E} \in \mathcal{A}$.

In order to be able to support such use-cases, this section now proceeds to present conditions for ensuring (6). This includes a sufficient and necessary condition of (6); however, to gradually introduce this condition to the reader, a condition is first presented that is only sufficient and not necessary, but that is less technical and directly follows from the necessary and sufficient condition.

4.1 Sufficient Condition for Preserving Satisfiability

Consider constructing a contract $(\mathcal{A}', \mathcal{G}')$ for component \mathbf{B}' such that (6) is ensured if \mathbf{B}' satisfies the contract. Since \mathcal{G}' is intended to express a property of the composition of \mathbf{B}' and a behavior $\mathbf{A} \in \mathcal{A}'$, and \mathcal{G} is to be implemented by the composition, it makes sense that $\mathcal{G}' = \mathcal{G}$. Regarding the assumption \mathcal{A}' , to enforce that the composition of \mathbf{B}' and any behavior implementing \mathcal{G} also implements \mathcal{G} , each behavior implementing \mathcal{G} should be in \mathcal{A}' . Since \mathcal{G} contains each behavior implementing it, it also makes sense that $\mathcal{A}' = \mathcal{G}$. Indeed, as shown in the following corollary, if \mathbf{B}' satisfies $(\mathcal{G}, \mathcal{G})$, then (6) holds.

Corollary 1. *Given a contract $(\mathcal{A}, \mathcal{G})$, if a behavior \mathbf{B}' satisfies $(\mathcal{G}, \mathcal{G})$, then*

$$\forall \mathbf{B}. \mathbf{B} \text{ satisfies } (\mathcal{A}, \mathcal{G}) \Rightarrow \mathbf{B}' \cap \mathbf{B} \text{ satisfies } (\mathcal{A}, \mathcal{G}).$$

Proof. This corollary follows from Theorem 1 in Sect. 4.2 since Definition 11 implies that $\mathcal{R}_{\mathcal{G}}(\mathcal{A}, \mathcal{G}) \subseteq \mathcal{G}$, which means, in accordance with (3), that: \mathbf{B}' satisfies $(\mathcal{G}, \mathcal{G}) \Rightarrow \mathbf{B}'$ satisfies $(\mathcal{R}_{\mathcal{G}}(\mathcal{A}, \mathcal{G}), \mathcal{G})$.

Example 7. Let $\Xi = \{x, y\}$ where x and y are Boolean variables and $T^\omega = \{0\}$. Consider a contract $(\mathcal{A}, \mathcal{G})$ where $\mathcal{A} = \{\mathbf{A}_1, \mathbf{A}_2\} = \{\{(0, 0)\}, \{(0, 0), (0, 1)\}\}$ and $\mathcal{G} = \{\mathbf{G}_1, \mathbf{G}_2, \mathbf{G}_3\} = \{\{(0, 0)\}, \{(1, 0)\}, \{(0, 0), (1, 0)\}\}$.

Behavior $\mathbf{B} = \{(0, 0), (1, 1)\}$ satisfies the contract $(\mathcal{A}, \mathcal{G})$ since

$$\mathbf{A}_1 \cap \mathbf{B} = \{(0, 0)\} \cap \{(0, 0), (1, 1)\} = \{(0, 0)\} = \mathbf{G}_1 \in \mathcal{G}$$

$$\mathbf{A}_2 \cap \mathbf{B} = \{(0, 0), (0, 1)\} \cap \{(0, 0), (1, 1)\} = \{(0, 0)\} = \mathbf{G}_1 \in \mathcal{G}.$$

Behavior $B'_1 = \{(0, 0), (1, 0), (1, 1)\}$ satisfies the contract $(\mathcal{G}, \mathcal{G})$ since

$$G_1 \cap B'_1 = \{(0, 0)\} \cap \{(0, 0), (1, 0), (1, 1)\} = \{(0, 0)\} = G_1 \in \mathcal{G}$$

$$G_2 \cap B'_1 = \{(1, 0)\} \cap \{(0, 0), (1, 0), (1, 1)\} = \{(1, 0)\} = G_2 \in \mathcal{G}$$

$$G_3 \cap B'_1 = \{(0, 0), (1, 0)\} \cap \{(0, 0), (1, 0), (1, 1)\} = \{(0, 0), (1, 0)\} = G_3 \in \mathcal{G}.$$

Then, in agreement with Corollary 1, composition $B'_1 \cap B$ satisfies $(\mathcal{A}, \mathcal{G})$ since

$$A_1 \cap B'_1 \cap B = \{(0, 0)\} \cap \{(0, 0), (1, 0), (1, 1)\} \cap \{(0, 0), (1, 1)\} = \{(0, 0)\} = G_1 \in \mathcal{G}$$

$$A_2 \cap B'_1 \cap B = \{(0, 0), (0, 1)\} \cap \{(0, 0), (1, 0), (1, 1)\} \cap \{(0, 0), (1, 1)\} = \{(0, 0)\} = G_1 \in \mathcal{G}.$$

On the other hand, the behavior $B'_2 = \{(0, 1), (1, 0), (1, 1)\}$ does not satisfy the contract $(\mathcal{G}, \mathcal{G})$ since

$$G_1 \cap B'_2 = \{(0, 0)\} \cap \{(0, 1), (1, 0), (1, 1)\} = \emptyset \notin \mathcal{G}.$$

Also so we see that the composition $B'_2 \cap B$ does not satisfy $(\mathcal{A}, \mathcal{G})$ since

$$A_1 \cap B'_2 \cap B = \{(0, 0)\} \cap \{(0, 1), (1, 0), (1, 1)\} \cap \{(0, 0), (1, 1)\} = \emptyset \notin \mathcal{G}.$$

□

4.2 Necessary and Sufficient Condition for Preserving Satisfiability

Section 4.1 presented Corollary 1 expressing a condition for ensuring (6). As previously mentioned, this condition is a sufficient, but not necessary condition of (6).

This is due to the fact that the assumption in the contract $(\mathcal{G}, \mathcal{G})$ may include behaviors that cannot be a composition of a behavior satisfying $(\mathcal{A}, \mathcal{G})$ and a behavior $A \in \mathcal{A}$. The subset of \mathcal{G} obtained by removing such behaviors from \mathcal{G} is here called the *realizable guarantee of $(\mathcal{A}, \mathcal{G})$* .

Definition 11 (Realizable Guarantee of Contract). *The realizable guarantee of a contract $(\mathcal{A}, \mathcal{G})$, denoted $\mathcal{R}_{\mathcal{G}}(\mathcal{A}, \mathcal{G})$, is the specification $\{A \cap B \mid A \in \mathcal{A} \wedge B \text{ satisfies } (\mathcal{A}, \mathcal{G})\}$.* □

As expressed in the following proposition, contracts $(\mathcal{A}, \mathcal{G})$ and $(\mathcal{A}, \mathcal{R}_{\mathcal{G}}(\mathcal{A}, \mathcal{G}))$ are satisfied by the exact same behaviors.

Proposition 2. *A behavior satisfies a contract $(\mathcal{A}, \mathcal{G})$ if and only if the behavior satisfies $(\mathcal{A}, \mathcal{R}_{\mathcal{G}}(\mathcal{A}, \mathcal{G}))$.*

Proof. Follows directly from (3) and Definition 11.

Having B' satisfy $(\mathcal{R}_{\mathcal{G}}(\mathcal{A}, \mathcal{G}), \mathcal{G})$ instead of $(\mathcal{G}, \mathcal{G})$ is indeed sufficient for ensuring (6). The behaviors in $\mathcal{G} \setminus \mathcal{R}_{\mathcal{G}}(\mathcal{A}, \mathcal{G})$ are not compositions of a behavior B satisfying $(\mathcal{A}, \mathcal{G})$ and a behavior $A \in \mathcal{A}$; therefore, these behaviors can be excluded from the assumption of $(\mathcal{G}, \mathcal{G})$ since the implication in (6) is trivially true for any behavior B that does not satisfy $(\mathcal{A}, \mathcal{G})$. The following theorem not only shows that it is indeed the case that B' satisfying $(\mathcal{R}_{\mathcal{G}}(\mathcal{A}, \mathcal{G}), \mathcal{G})$ is sufficient for ensuring (6), but that this is in fact also a necessary condition.

Theorem 1. *Given a contract $(\mathcal{A}, \mathcal{G})$, a behavior B' satisfies $(\mathcal{R}_{\mathcal{G}}(\mathcal{A}, \mathcal{G}), \mathcal{G})$ if and only if:*

$$\forall B. B \text{ satisfies } (\mathcal{A}, \mathcal{G}) \Rightarrow B' \cap B \text{ satisfies } (\mathcal{A}, \mathcal{G}). \quad (7)$$

The proof of Theorem 1 follows after the following lemma.

Lemma 1. *A behavior satisfies a contract $(\mathcal{A}_1 \cup \dots \cup \mathcal{A}_N, \mathcal{G})$ if the behavior satisfies each contract $(\mathcal{A}_i, \mathcal{G})$.*

Proof of Lemma 1. In accordance with (3), consider as given a behavior B such that $\forall A_i \in \mathcal{A}_i. A_i \cap B \in \mathcal{G}$. It follows directly that $\forall A \in (\mathcal{A}_1 \cup \dots \cup \mathcal{A}_N). A \cap B \in \mathcal{G}$, which means that B satisfies $(\mathcal{A}_1 \cup \dots \cup \mathcal{A}_N, \mathcal{G})$ in accordance with (3).

Proof of Theorem 1. For the if-only case, in accordance with (3), consider as given

$$\forall A \in \mathcal{R}_{\mathcal{G}}(\mathcal{A}, \mathcal{G}). A \cap B' \in \mathcal{G}. \quad (8)$$

Consider an arbitrary behavior B that satisfies $(\mathcal{A}, \mathcal{G})$, which means that $\forall A \in \mathcal{A}. A \cap B \in \mathcal{R}_{\mathcal{G}}(\mathcal{A}, \mathcal{G})$ in accordance with Proposition 2 and (3). This and (8) imply $\forall A \in \mathcal{A}. A \cap B \cap B' \in \mathcal{G}$, which means that $B' \cap B$ satisfies $(\mathcal{A}, \mathcal{G})$ in accordance with (3). This also holds for each B satisfying $(\mathcal{A}, \mathcal{G})$ since B characterizes an arbitrary behavior, and (7) hence holds.

For the if case, consider that (7) holds. It follows in accordance with (3) that

$$\forall B. (B \text{ satisfies } (\mathcal{A}, \mathcal{G}) \Rightarrow (\forall A' \in \mathcal{A}. A' \cap B' \cap B \in \mathcal{G})).$$

Notably, $\forall A' \in \mathcal{A}. A' \cap B' \cap B \in \mathcal{G}$ can be rewritten as $\forall A \in \{A' \cap B \mid A' \in \mathcal{A}\}. A \cap B' \in \mathcal{G}$, which in accordance with (3) means that

$$\forall B. B \text{ satisfies } (\mathcal{A}, \mathcal{G}) \Rightarrow B' \text{ satisfies } (\{A \cap B \mid A \in \mathcal{A}\}, \mathcal{G}).$$

This and Lemma 1 imply that B' satisfies

$$\left(\bigcup_{B, B \text{ satisfies } (\mathcal{A}, \mathcal{G})} \{A \cap B \mid A \in \mathcal{A}\}, \mathcal{G} \right),$$

which is equivalent to B' satisfying $(\{A \cap B \mid A \in \mathcal{A}, B \text{ satisfies } (\mathcal{A}, \mathcal{G})\}, \mathcal{G})$. In accordance with Definition 11, this means that B' satisfies $(\mathcal{R}_{\mathcal{G}}(\mathcal{A}, \mathcal{G}), \mathcal{G})$, which ends the proof. \square

4.3 Preserving Satisfiability Through Interface Port Specifications

Similar to Sects. 4.1 and 4.2, this section presents a condition for preserving contract satisfiability; however, in contrast to the conditions presented in Sects. 4.1 and 4.2, rather than ensuring (6), this condition is sufficient for

$$\forall B \in \mathcal{I}_X. B \text{ satisfies } (\mathcal{A}, \mathcal{G}) \Rightarrow B' \cap B \text{ satisfies } (\mathcal{A}, \mathcal{G}), \quad (9)$$

where $\mathcal{I}_X = \{B \mid X_B \subseteq X\}$. In contrast to (6), which quantifies over each behavior, formula (9) quantifies over each behavior B that implements a special type of specification, here denoted \mathcal{I}_X and called *interface port specification*, which expresses only a restriction on the set of variables that B can constrain.

Definition 12 (Interface Port Specification). An interface port specification for X is a specification $\{\mathbf{B} \mid X_{\mathbf{B}} \subseteq X\}$. \square

Prior to presenting a theorem ensuring (9), an example will be studied.

Example 8. Consider $\Xi = \{x, y, u, v\}$ where x, y, u , and v are Boolean variables and $T^w = \{0\}$. Let \mathbf{B}' be a behavior and $(\mathcal{A}, \mathcal{G})$ a contract where:

- (a) $\mathbf{B}' = \{(x, y, u, v) \in \Omega \mid u = v\}$;
- (b) $\mathcal{G} = \{\mathbf{B} \neq \emptyset \mid \forall (x, y, u, v) \in \mathbf{B}. x = 1\}$; and
- (c) $\mathcal{A} = \{\mathbf{B} \mid \forall (x, y, u, v) \in \mathbf{B}. y = 1\} \cap \mathcal{I}_{\{x, y\}}$.

Consider a behavior $\mathbf{B} = \{(x, y, u, v) \in \Omega \mid x = y\}$, and an arbitrary behavior $\mathbf{A} \in \mathcal{A}$. Their intersection $\mathbf{A} \cap \mathbf{B}$ is behavior $\{(x, y, u, v) \in \Omega \mid x = y = 1\}$, which implements \mathcal{G} . This means that in order for (9) to hold, behavior $\mathbf{B}' \cap \mathbf{A} \cap \mathbf{B}$ must also implement \mathcal{G} . This is indeed the case since $\mathbf{B}' \cap \mathbf{A} \cap \mathbf{B} = \{(x, y, u, v) \in \Omega \mid x = y = 1 \text{ and } u = v\}$, contained in \mathcal{G} . \square

In accordance with Definitions 10 and 12, conditions (a)–(c) in Example 8 respectively imply:

- (i) $\mathbf{B}' \in \mathcal{I}_{\{u, v\}}$;
- (ii) \mathcal{G} is agnostic to $\{u, v, y\} \supseteq \{u, v\}$ and $\emptyset \notin \mathcal{G}$; and
- (iii) $\mathcal{A} \subseteq \mathcal{I}_{\{u, v\}}^c$.

As will be shown in the following theorem, conditions (i)–(iii) ensure that if a behavior \mathbf{B} satisfies $(\mathcal{A}, \mathcal{G})$ and implements an interface specification for a set disjoint to $\{u, v\}$, e.g. $\{x, y\}$ as in Example 8, then it holds that $\mathbf{B}' \cap \mathbf{B}$ satisfies $(\mathcal{A}, \mathcal{G})$.

Theorem 2. Given a contract $(\mathcal{A}, \mathcal{G})$ and two disjoint sets of variables X and Y , if

- (i) $\mathbf{B}' \in \mathcal{I}_Y$
- (ii) \mathcal{G} is agnostic to a set $Z \supseteq Y$ and $\emptyset \notin \mathcal{G}$; and
- (iii) $\mathcal{A} \subseteq \mathcal{I}_{Y^c}$,

then it holds: $\forall \mathbf{B} \in \mathcal{I}_X. \mathbf{B} \text{ satisfies } (\mathcal{A}, \mathcal{G}) \Rightarrow \mathbf{B}' \cap \mathbf{B} \text{ satisfies } (\mathcal{A}, \mathcal{G})$.

The lemmas used in the following proof can be found in Appendix A.

Proof. Assume (i)–(iii), and an arbitrary behavior $\mathbf{B} \in \mathcal{I}_X$ satisfying $(\mathcal{A}, \mathcal{G})$, i.e., $\forall \mathbf{A} \in \mathcal{A}. \mathbf{A} \cap \mathbf{B} \in \mathcal{G}$ in accordance with (3). From (i), in accordance with Definition 12, it follows that $X_{\mathbf{B}'} \subseteq Y$. In accordance with Lemma 4, this and (ii) imply that \mathcal{G} is agnostic to $X_{\mathbf{B}'}$. For an arbitrary $\mathbf{A} \in \mathcal{A}$, in accordance with Definition 10 and since $\mathbf{A} \cap \mathbf{B} \in \mathcal{G}$, it follows that

$$\left(\text{proj}_{X_{\mathbf{B}'}}^c(\mathbf{B}' \cap (\mathbf{A} \cap \mathbf{B})) = \text{proj}_{X_{\mathbf{B}'}}^c(\mathbf{A} \cap \mathbf{B}) \right) \Rightarrow \mathbf{B}' \cap (\mathbf{A} \cap \mathbf{B}) \in \mathcal{G}. \quad (10)$$

In accordance with Definition 12, $\mathbf{B} \in \mathcal{I}_X$ and (iii) imply $X_{\mathbf{B}} \subseteq X$ and $X_{\mathbf{A}} \subseteq Y^c$, respectively. Since $X \cap Y = \emptyset$ was given, it follows that $X_{\mathbf{B}} \subseteq Y^c$, and further

that $X_A \cup X_B \subseteq Y^C$. Then in accordance with Lemma 5 and since $A \cap B \neq \emptyset$ due to $\emptyset \notin \mathcal{G}$ in (ii), it holds that $X_{A \cap B} \subseteq Y^C$, which means that $X_{A \cap B} \cap X_{B'} = \emptyset$ since $X_{B'} \subseteq Y$. In accordance with Proposition 1, this means that (10) is equivalent to

$$\left(\text{proj}_{X_{B'}^c}(B') \cap \text{proj}_{X_{B'}^c}(A \cap B) = \text{proj}_{X_{B'}^c}(A \cap B) \right) \Rightarrow B' \cap (A \cap B) \in \mathcal{G}. \quad (11)$$

In accordance with Lemma 2, it holds that $\text{proj}_{X_{B'}^c}(B') = \Omega$, which means that the left side of the implication in (11) is equivalent to $\Omega \cap \text{proj}_{X_{B'}^c}(A \cap B) = \text{proj}_{X_{B'}^c}(A \cap B)$. Since Ω includes all runs, it follows that $\text{proj}_{X_{B'}^c}(A \cap B) = \text{proj}_{X_{B'}^c}(A \cap B)$, which means that it follows that the right side of the implication in (11) holds, i.e., $B' \cap (A \cap B) \in \mathcal{G}$. Since A characterizes an arbitrary behavior in \mathcal{A} , in accordance with (3), $B' \cap B$ satisfies $(\mathcal{A}, \mathcal{G})$. Since B characterizes an arbitrary behavior $B \in \mathcal{I}_X$ satisfying $(\mathcal{A}, \mathcal{G})$, it follows that $\forall B \in \mathcal{I}_X. B \text{ satisfies } (\mathcal{A}, \mathcal{G}) \Rightarrow B' \cap B \text{ satisfies } (\mathcal{A}, \mathcal{G})$, which ends the proof. \square

Example 9. Consider contract $(\mathcal{A}, \mathcal{G})$ and behavior B' from Example 8. As previously mentioned, it holds that: (i) $B' \in \mathcal{I}_{\{u,v\}}$; (ii) \mathcal{G} is agnostic to $\{u, v, y\} \supseteq \{u, v\}$ and $\emptyset \notin \mathcal{G}$; and (iii) $\mathcal{A} \subseteq \mathcal{I}_{\{u,v\}^c}$. In accordance with Theorem 2, to establish that the composition of B' and a behavior B satisfies contract $(\mathcal{A}, \mathcal{G})$, it suffices to show that B satisfies $(\mathcal{A}, \mathcal{G})$ and implements an interface specification for a subset of $\{x, y\}$. An example of such a behavior is $B = \{(x, y, u, v) \in \Omega \mid x = y\}$, presented in Example 8. \square

5 Conclusion

As argued for, and exemplified in Sect. 1, to support properties that may be invalidated through composition, a contracts theory should not have composition inherently monotonic with respect to implementation. Therefore, the paper has presented, as a first contribution, the basis for a novel contracts theory where composition is non-monotonic with respect to implementation. The key solution is that specifications are represented as sets of sets of runs instead of only sets of runs as in previous theories.

The general support from non-monotony implies that cases where monotony is indeed desired needs to be managed explicitly. To support this, the paper has, as a second contribution, presented Corollary 1 and Theorems 1 and 2, which each provides conditions to ensure monotony of composition with respect to implementation when indeed desired.

A Propositions and Lemmas

The following two lemmas are used to prove Propositions 1 and 2, which follow after the lemmas.

Lemma 2. *For any two sets of variables X and Y , it holds that*

$$\text{proj}_Y(\text{proj}_X \mathbf{B}) = \text{proj}_{X \cap Y} \mathbf{B}.$$

Proof. Consider an arbitrary $\omega \in \text{proj}_Y(\text{proj}_X \mathbf{B})$. In accordance with Definition 3, this means that $\exists \mu. \omega[Y] = \mu[Y] \wedge \mu \in \{\omega' \in \Omega \mid \omega'[X] = \nu[X], \nu \in \mathbf{B}\}$. This can be rewritten as $\exists \nu \exists \mu. \omega[Y] = \mu[Y] \wedge \mu[X] = \nu[X] \wedge \nu \in \mathbf{B}$. Since $X \cap Y \subseteq X$ and $X \cap Y \subseteq Y$, and $\mu[X] = \nu[X]$ and $\omega[Y] = \mu[Y]$, it holds that $\mu[X \cap Y] = \nu[X \cap Y]$ and $\omega[X \cap Y] = \mu[X \cap Y]$, which also means that $\omega[X \cap Y] = \nu[X \cap Y]$. Thus, it holds $\exists \nu. \omega[X \cap Y] = \nu[X \cap Y] \wedge \nu \in \mathbf{B}$. In accordance with Definition 3, this means that $\omega \in \text{proj}_{X \cap Y} \mathbf{B}$. Since ω represents an arbitrary run in $\text{proj}_Y(\text{proj}_X \mathbf{B})$, it must hold that each run in $\text{proj}_Y(\text{proj}_X \mathbf{B})$ is also in $\text{proj}_{X \cap Y} \mathbf{B}$, which means that $\text{proj}_Y(\text{proj}_X \mathbf{B}) \subseteq \text{proj}_{X \cap Y} \mathbf{B}$.

Next consider an arbitrary $\omega \in \text{proj}_{X \cap Y} \mathbf{B}$. In accordance with Definition 3, this means that $\exists \nu. \omega[X \cap Y] = \nu[X \cap Y] \wedge \nu \in \mathbf{B}$. Create a run μ such that $\mu[X] = \nu[X]$ and $\mu[X^c] = \omega[X^c]$. Since $\mu[X^c] = \omega[X^c]$, $\omega[X \cap Y] = \nu[X \cap Y]$, and $Y \subseteq X^c \cup (X \cap Y)$, it follows that $\omega[Y] = \mu[Y]$. Thus, we have shown that $\exists \nu \exists \mu. \omega[Y] = \mu[Y] \wedge \mu[X] = \nu[X] \wedge \nu \in \mathbf{B}$, which can, as previously mentioned, be rewritten as $\exists \mu. \omega[Y] = \mu[Y] \wedge \mu \in \{\omega' \in \Omega \mid \omega'[X] = \nu[X], \nu \in \mathbf{B}\}$. In accordance with Definition 3, this means that $\omega \in \text{proj}_Y(\text{proj}_X \mathbf{B})$. Since ω represents an arbitrary run in $\text{proj}_{X \cap Y} \mathbf{B}$, it must hold that each run in $\text{proj}_{X \cap Y} \mathbf{B}$ is also in $\text{proj}_Y(\text{proj}_X \mathbf{B})$, which means that $\text{proj}_{X \cap Y} \mathbf{B} \subseteq \text{proj}_Y(\text{proj}_X \mathbf{B})$. Since it was previously also shown that $\text{proj}_Y(\text{proj}_X \mathbf{B}) \subseteq \text{proj}_{X \cap Y} \mathbf{B}$, it follows that $\text{proj}_Y(\text{proj}_X \mathbf{B}) = \text{proj}_{X \cap Y} \mathbf{B}$, which ends the proof. \square

Lemma 3. *It holds that $\text{proj}_{X_{\mathbf{B}}} \mathbf{B} = \mathbf{B}$.*

Proof. In accordance with Definition 4, it holds that $\text{proj}_{\{x\}^c} \mathbf{B} = \mathbf{B}$ for each variable in $X_{\mathbf{B}}^c = \{x_1, \dots, x_N\}$. It follows that

$$\text{proj}_{\{x_1\}^c}(\text{proj}_{\{x_2\}^c}(\dots \text{proj}_{\{x_N\}^c}(\mathbf{B}) \dots)) = \mathbf{B}.$$

It follows in accordance with Lemma 2 that $\text{proj}_{\{x_1\}^c \cap \dots \cap \{x_N\}^c} \mathbf{B} = \mathbf{B}$ and since $X_{\mathbf{B}} = \{x_1\}^c \cap \dots \cap \{x_N\}^c$, it also holds that $\text{proj}_{X_{\mathbf{B}}} \mathbf{B} = \mathbf{B}$, which ends the proof. \square

Proposition 1. *Let \mathbf{B}_1 and \mathbf{B}_2 be two behaviors such that $X_{\mathbf{B}_1} \cap X_{\mathbf{B}_2} = \emptyset$. Then, for an arbitrary set of variables Y , it holds*

$$\text{proj}_Y(\mathbf{B}_1 \cap \mathbf{B}_2) = \text{proj}_Y \mathbf{B}_1 \cap \text{proj}_Y \mathbf{B}_2.$$

Proof. To prove $\text{proj}_Y(\mathbf{B}_1 \cap \mathbf{B}_2) \subseteq \text{proj}_Y \mathbf{B}_1 \cap \text{proj}_Y \mathbf{B}_2$, assume $\omega \in \text{proj}_Y(\mathbf{B}_1 \cap \mathbf{B}_2)$. According to Definition 3, this means that there exist a $\nu \in (\mathbf{B}_1 \cap \mathbf{B}_2)$ such that $\nu[Y] = \omega[Y]$. Thus for $i = 1, 2$, it holds that $\nu \in \mathbf{B}_i$ and $\nu[Y] = \omega[Y]$, and therefore that $\omega \in \text{proj}_Y \mathbf{B}_i$, which ends the proof.

To prove also that $\text{proj}_Y \mathbf{B}_1 \cap \text{proj}_Y \mathbf{B}_2 \subseteq \text{proj}_Y(\mathbf{B}_1 \cap \mathbf{B}_2)$, consider an arbitrary run $\omega \in \text{proj}_Y \mathbf{B}_1 \cap \text{proj}_Y \mathbf{B}_2$. According to Definition 3 this means that:

$$\exists \nu_1 \in \mathbf{B}_1. \nu_1[Y] = \omega[Y]$$

$$\exists \nu_2 \in \mathbf{B}_2. \nu_2[Y] = \omega[Y]$$

Now consider the vector $\nu = (\nu_1[X_{B_1}], \nu_2[X_{B_1}^C])$. Clearly it holds that $\omega[Y] = \nu[Y]$.

Since $\nu_1 \in B_1$, and $\nu[X_{B_1}] = \nu_1[X_{B_1}]$, it holds according to Definition 3 that $\nu \in \text{proj}_{X_{B_1}} B_1$. Then, according to Lemma 3, it also holds that $\nu_1 \in B_1$. Similarly, since $\nu_2 \in B_2$, and $\nu[X_{B_2}] = \nu_2[X_{B_2}]$, it must hold that $\nu \in B_2$.

Thus, we have shown that $\nu \in B_1 \cap B_2$ and that $\omega[Y] = \nu[Y]$. Therefore, according to Definition 3, it holds that $\omega \in \text{proj}_Y(B_1 \cap B_2)$, which ends the proof. \square

Proposition 2. *It holds that $\text{proj}_Y B = \Omega$ if $Y \cap X_B \neq \emptyset$.*

Proof. In accordance with Lemma 3, it holds that $\text{proj}_Y B = \text{proj}_Y(\text{proj}_{X_B} B)$, and further that $\text{proj}_Y B = \text{proj}_{Y \cap X_B} B$ in accordance with Lemma 2. If $Y \cap X_B \neq \emptyset$, it follows that $\text{proj}_Y B = \text{proj}_{\emptyset} B = \Omega$. \square

The following two lemmas are used to prove Theorem 2.

Lemma 4. *If a behavior set \mathcal{Q} is agnostic to a variable set X , then it is also agnostic to any variable set $Y \subseteq X$.*

Proof. Consider a $B \in \mathcal{Q}$ and a B' such that $\text{proj}_{Y^C} B' = \text{proj}_{Y^C} B$. Since $X^C \subseteq Y^C$, it holds according to Lemma 2, that

$$\text{proj}_{X^C} B' = \text{proj}_{X^C} \text{proj}_{Y^C} B' = \text{proj}_{X^C} \text{proj}_{Y^C} B = \text{proj}_{X^C} B$$

From this equality and since \mathcal{Q} is agnostic to X , it follows that $B' \in \mathcal{Q}$. Thus, we have shown that \mathcal{Q} is agnostic to also Y . \square

Lemma 5. *Given two behaviors B and B' where $B \cap B' \neq \emptyset$, it holds that $X_{B \cap B'} \subseteq X_B \cup X_{B'}$.*

Proof. Assume $X_{B \cap B'} \not\subseteq X_B \cup X_{B'}$, which will be shown to lead to a contradiction. This means that there exists an x such that $x \notin X_B$, $x \notin X_{B'}$, and $x \in X_{B \cap B'}$. In accordance with Definition 4, this respectively means that

$$B = \text{proj}_{\{x\}^C} B \tag{12}$$

$$B' = \text{proj}_{\{x\}^C} B', \text{ and} \tag{13}$$

$$B \cap B' \neq \text{proj}_{\{x\}^C}(B \cap B'). \tag{14}$$

Proposition 1 and (14) imply $B \cap B' \neq \text{proj}_{\{x\}^C} B \cap \text{proj}_{\{x\}^C} B'$, which in combination with (12) and (13) imply that $B \cap B' \neq B \cap B'$. Since this is a contradiction, assumption $X_{B \cap B'} \not\subseteq X_B \cup X_{B'}$ must be false, and it must rather hold that $X_{B \cap B'} \subseteq X_B \cup X_{B'}$, which completes the proof. \square

References

1. Bauer, S.S., David, A., Hennicker, R., Guldstrand Larsen, K., Legay, A., Nyman, U., Wasowski, A.: Moving from specifications to contracts in component-based design. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 43–58. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28872-2_3
2. Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple viewpoint contract-based specification and design. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 200–225. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-92188-2_9
3. Benveniste, A., Caillaud, B., Nickovic, D., Passerone, R., Raclet, J.B., Reinkemeier, P., Sangiovanni-Vincentelli, A., Damm, W., Henzinger, T., Larsen, K.G.: Contracts for system design. Rapport de recherche RR-8147, INRIA, November 2012. <http://hal.inria.fr/hal-00757488>
4. Benveniste, A., Caillaud, B., Passerone, R.: Multi-viewpoint state machines for rich component models. In: Nicolescu, G., Mosterman, P. (eds.) Model-Based Design for Embedded Systems, pp. 487–518. Taylor & Francis (2009). <http://www.google.se/books?id=8Cjg2mM-m1MC>
5. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. *J. ACM* **31**(3), 560–599 (1984). <http://doi.acm.org/10.1145/828.833>
6. Cimatti, A., Tonetta, S.: A property-based proof system for contract-based design. In: 2012 38th Euromicro Conference on Software Engineering and Advanced Applications, pp. 21–28, September 2012
7. Dill, D.L.: Trace theory for automatic hierarchical verification of speed-independent circuits. In: Proceedings of the Fifth MIT Conference on Advanced Research in VLSI, pp. 51–65. MIT Press, Cambridge (1988). <http://dl.acm.org/citation.cfm?id=88056.88061>
8. Hoare, C.A.R.: An axiomatic basis for computer programming. *Commun. ACM* **12**(10), 576–580 (1969). <http://doi.acm.org/10.1145/363235.363259>
9. Maier, P.: A set-theoretic framework for assume-guarantee reasoning. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 821–834. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-48224-5_67
10. Meyer, B.: Applying “Design by Contract”. *IEEE Comput.* **25**, 40–51 (1992)
11. Negulescu, R.: Process spaces. In: Proceedings of the 11th International Conference on Concurrency Theory, CONCUR 2000, pp. 199–213. Springer, London (2000). <http://dl.acm.org/citation.cfm?id=646735.701627>
12. Quinton, S., Graf, S.: Contract-based verification of hierarchical systems of components. In: Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, pp. 377–381, November 2008
13. Westman, J., Nyberg, M.: Conditions of contracts for separating responsibilities in heterogeneous systems. *Form. Methods Syst. Des.* **52**(2), 147–192 (2017). <https://doi.org/10.1007/s10703-017-0294-7>
14. Wolf, E.S.: Hierarchical models of synchronous circuits for formal verification and substitution. Ph.D. thesis. Stanford University, Stanford, CA, USA (1996). uMI Order No. GAX96-12052



Correction to: Applied Choreographies

Saverio Giallorenzo, Fabrizio Montesi, and Maurizio Gabbrielli

Correction to:
Chapter “Applied Choreographies”
in: C. Baier and L. Caires (Eds.): *Formal Techniques*
***for Distributed Objects, Components, and Systems*, LNCS 10854,**
https://doi.org/10.1007/978-3-319-92612-4_2

In the originally published version of this chapter the acknowledgment was missing. This has been added.

The updated version of this chapter can be found at
https://doi.org/10.1007/978-3-319-92612-4_2

© IFIP International Federation for Information Processing 2018
Published by Springer International Publishing AG 2018. All Rights Reserved
C. Baier and L. Caires (Eds.): FORTE 2018, LNCS 10854, p. E1, 2018.
https://doi.org/10.1007/978-3-319-92612-4_11

Author Index

Akili, Samira 161
Alrahman, Yehia Abd 1

De Nicola, Rocco 1

Gabbrielli, Maurizio 21
Garbi, Giulio 1
Giallorenzo, Saverio 21
Girault, Alain 41
Gössler, Gregor 41
Guerraoui, Rachid 41

Hamza, Jad 41
Hedin, Daniel 141

Lanotte, Ruggero 58
Loreti, Michele 1

Merro, Massimo 58
Montesi, Fabrizio 21
Munteanu, Andrei 58

Nantes, Daniele 79
Nyberg, Mattias 181

Pantović, Jovanka 101
Pérez, Jorge A. 79
Petrucci, Laure 121
Prokić, Ivan 101

Sabelfeld, Andrei 141
Seredinschi, Dragos-Adrian 41
Sjösten, Alexander 141

van de Pol, Jaco 121
Vieira, Hugo Torres 101
Völlinger, Kim 161

Westman, Jonas 181