

Gwen Salaün
Anton Wijs (Eds.)

LNCS 13077

Formal Aspects of Component Software

17th International Conference, FACS 2021
Virtual Event, October 28–29, 2021
Proceedings



Springer

Founding Editors

Gerhard Goos

Karlsruhe Institute of Technology, Karlsruhe, Germany

Juris Hartmanis

Cornell University, Ithaca, NY, USA

Editorial Board Members

Elisa Bertino

Purdue University, West Lafayette, IN, USA

Wen Gao

Peking University, Beijing, China

Bernhard Steffen 

TU Dortmund University, Dortmund, Germany

Gerhard Woeginger 

RWTH Aachen, Aachen, Germany

Moti Yung 

Columbia University, New York, NY, USA

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

Gwen Salaün · Anton Wijs (Eds.)

Formal Aspects of Component Software

17th International Conference, FACS 2021
Virtual Event, October 28–29, 2021
Proceedings

Editors

Gwen Salaün
Grenoble Alpes University
Saint-Martin-d'Hères, France

Anton Wijs 
Eindhoven University of Technology
Eindhoven, The Netherlands

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-030-90635-1 ISBN 978-3-030-90636-8 (eBook)
<https://doi.org/10.1007/978-3-030-90636-8>

LNCS Sublibrary: SL2 – Programming and Software Engineering

© Springer Nature Switzerland AG 2021

Chapter “A Linear Parallel Algorithm to Compute Bisimulation and Relational Coarsest Partitions” is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>). For further details see license information in the chapter.

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, expressed 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.

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

Preface

This volume contains the proceedings of the 17th International Conference on Formal Aspects of Component Software (FACS 2021), held online, due to the COVID-19 pandemic, during October 28–29, 2021.

Component-based software development proposes sound engineering principles and techniques to cope with the complexity of present-day software systems. However, many challenging conceptual and technological issues remain in component-based software development theory and practice. Furthermore, the advent of service-oriented and cloud computing, cyber-physical systems, and the Internet of Things has brought to the fore new dimensions, such as quality of service and robustness to withstand faults, which require revisiting established concepts and developing new ones.

FACS 2021 was concerned with how formal methods can be applied to component-based software and system development. Formal methods have provided foundations for component-based software through research on mathematical models for components, composition and adaptation, and rigorous approaches to verification, deployment, testing, and certification.

We received 16 submissions for the conference, and all of them were reviewed by three reviewers. Based on their reports and subsequent discussions, the Program Committee (PC) decided to accept eight papers (seven regular papers and one tool paper) for inclusion in this volume and the program of FACS 2021. In addition, we invited Radu Calinescu and Corina Pasareanu to give keynotes. This volume contains an abstract of the talk given by Radu Calinescu and an invited paper by Corina Pasareanu.

We thank Radu Calinescu and Corina Pasareanu for accepting our invitations to give an invited talk, as well as all authors who submitted their work for FACS 2021. We thank the members of the PC for their effort to write timely and high-quality reviews, and their discussions to make the final selection of papers. We also thank the FACS Steering Committee for useful suggestions and support. Finally, we thank the other members of the FACS 2021 organizing committee, Radu Mateescu, Ajay Muroor Nadumane, and Ahang Zuo, for their contribution to organizing the conference.

September 2021


Gwen Salaün
Anton Wijs

Peter Csaba Ölveczky	University of Oslo, Norway
Jun Pang	University of Luxembourg, Luxembourg
José Proença	CISTER, Portugal
Jorge Pérez	University of Groningen, The Netherlands
Camilo Rocha	Pontificia Universidad Javeriana Cali, Colombia
Gwen Salaün	Université Grenoble Alpes, France
Ana Sokolova	University of Salzburg, Austria
Jacopo Soldani	University of Pisa, Italy
Anton Wijs	Eindhoven University of Technology, The Netherlands
Shoji Yuen	Nagoya University, Japan

Additional Reviewers

Zahra Moezkarimi	Zeynab Sabahi Kaviani
Yuanrui Zhang	Wanwei Liu
Vincent Hugot	Renato Neves

Parametric and Interval Model Checking: Recent Advances and Applications (Abstract of Invited Paper)

Radu Calinescu 

Department of Computer Science, University of York, UK
radu.calinescu@york.ac.uk

Abstract. The model checking of Markov chains is a powerful technique for verifying performance, dependability and other key properties of systems with stochastic behaviour, both during development and at runtime. However, the usefulness of this technique depends on the accuracy of the models being verified, and on the efficiency of the verification. This invited talk will describe how recent advances in parametric and interval model checking address major challenges posed by these prerequisites, enabling the application of the technique to a broader range of component-based systems.

Keywords: Parametric model checking · Parametric Markov chains · Confidence-interval model checking · Interval Markov chains · Change-point detection

This talk is based on research reported in [1–7], and funded by the UK Research and Innovation project EP/V026747/1 ‘Trustworthy Autonomous Systems Node in Resilience’, the Assuring Autonomy International Programme, and the ORCA-Hub Partnership Resource Fund project ‘COVE’.

References

1. Alasmari, N., Calinescu, R., Paterson, C., Mirandola, R.: Quantitative verification with adaptive uncertainty reduction. arXiv preprint arXiv: <https://arxiv.org/abs/2109.02984> (2021)
2. Calinescu, R., Ceska, M., Gerasimou, S., Kwiatkowska, M., Paoletti, N.: Efficient synthesis of robust models for stochastic systems. *J. Syst. Softw.* **143**, 140–158 (2018). <https://doi.org/10.1016/j.jss.2018.05.013>
3. Calinescu, R., Ghezzi, C., Johnson, K., Pezzé, M., Rafiq, Y., Tamburrelli, G.: Formal verification with confidence intervals to establish quality of service properties of software systems. *IEEE Trans. Reliab.* **65**(1), 107–125 (2016). <https://doi.org/10.1109/TR.2015.2452931>
4. Calinescu, R., Johnson, K., Paterson, C.: FACT: A probabilistic model checker for formal verification with confidence intervals. In: 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 540–546 (2016). https://doi.org/10.1007/978-3-662-49674-9_32

5. Calinescu, R., Paterson, C., Johnson, K.: Efficient parametric model checking using domain knowledge. *IEEE Trans. Softw. Eng.* **47**(6), 1114–1133 (2021). <https://doi.org/10.1109/TSE.2019.2912958>
6. Fang, X., Calinescu, R., Gerasimou, S., Alhwikem, F.: Fast parametric model checking through model fragmentation. In: 43rd IEEE/ACM International Conference on Software Engineering (ICSE), pp. 835–846 (2021). <https://doi.org/10.1109/ICSE43902.2021.00081>
7. Zhao, X., Calinescu, R., Gerasimou, S., Robu, V., Flynn, D.: Interval change-point detection for runtime probabilistic model checking. In: 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 163–174 (2020). <https://doi.org/10.1145/3324884.3416565>

Contents

Invited Paper

Learning Assumptions for Verifying Cryptographic Protocols Compositionally	3
<i>Zichao Zhang, Arthur Azevedo de Amorim, Limin Jia, and Corina Păsăreanu</i>	

Modelling and Composition

Component-Based Approach Combining UML and BIP for Rigorous System Design	27
<i>Salim Chehida, Abdelhakim Baouya, and Saddek Bensalem</i>	
Composable Partial Multiparty Session Types	44
<i>Claude Stolze, Marino Miculan, and Pietro Di Gianantonio</i>	
A Canonical Algebra of Open Transition Systems	63
<i>Elena Di Lavore, Alessandro Gianola, Mario Román, Nicoletta Sabadini, and Paweł Sobociński</i>	
Corinne, a Tool for Choreography Automata	82
<i>Simone Orlando, Vairo Di Pasquale, Franco Barbanera, Ivan Lanese, and Emilio Tuosto</i>	

Verification

Specification and Safety Verification of Parametric Hierarchical Distributed Systems	95
<i>Marius Bozga and Radu Iosif</i>	
A Linear Parallel Algorithm to Compute Bisimulation and Relational Coarsest Partitions	115
<i>Jan Martens, Jan Friso Groote, Lars van den Haak, Pieter Hijma, and Anton Wijs</i>	
Automated Generation of Initial Configurations for Testing Component Systems	134
<i>Frédéric Dadeau, Jean-Philippe Gros, and Olga Kouchnarenko</i>	

Monitoring Distributed Component-Based Systems 153
Yliès Falcone, Hosein Nazarpour, Saddek Bensalem, and Marius Bozga

Author Index 175

Invited Paper



Learning Assumptions for Verifying Cryptographic Protocols Compositionally

Zichao Zhang¹, Arthur Azevedo de Amorim³, Limin Jia¹,
and Corina Păsăreanu^{1,2}(✉)

¹ Carnegie Mellon University, Pittsburgh, USA

{zichaoz, liminjia}@andrew.cmu.edu

² NASA Ames, Mountain View, USA

pcorina@cmu.edu

³ Boston University, Boston, USA

Abstract. Automated analysis tools for cryptographic protocols can verify sophisticated designs, but lack compositionality. To address this limitation, we investigate the use of automata learning for verifying authentication protocols in an automatic and compositional way. We present *Taglierino*, a tool for synthesizing specifications for protocol components and checking them in isolation. The specifications can aid the design of protocol variants and speed up verification. *Taglierino* includes a domain-specific language for protocols, which are compiled to automata and analyzed with the LTSA model checker extended with automata learning. We demonstrate the tool on a series of case studies, including the Needham-Schroeder, Woo-Lam, and Kerberos protocols.

Keywords: Assume-guarantee reasoning · Automata learning · Protocols

1 Introduction

Cryptographic protocols such as TLS are crucial for security, but notoriously difficult to get right. Automated analyses [14, 15, 53] can help discover vulnerabilities in sophisticated designs before deployment [10, 12, 18, 37, 54], and are thus invaluable to protocol development. Unfortunately, they suffer from a key drawback: limited compositional reasoning. To verify a property, they must analyze the entire protocol at once, rather than verifying its components against separate specifications. This is unsatisfactory for several reasons. First, decomposition can speed up verification, since it reduces the code analyzed in each stage. Second, a monolithic analysis provides few guarantees when the protocol is itself part of a larger system—e.g. using a key to sign and encrypt data simultaneously can enable attacks that are absent if only one of the functionalities is used. Finally, decomposition can guide protocol design, helping to find modifications that are

A. A. de Amorim—Work performed at Carnegie Mellon University.

© Springer Nature Switzerland AG 2021

G. Salaün and A. Wijs (Eds.): FACS 2021, LNCS 13077, pp. 3–23, 2021.

https://doi.org/10.1007/978-3-030-90636-8_1

also secure. Sadly, prior work on compositional protocol verification [27] requires significant manual effort to find component assumptions.

We envision a future where the power of compositionality can coexist with the convenience of automation. As a first step, we consider how protocol analysis can benefit from off-the-shelf, automated compositional techniques, by developing *Taglierino*, a verification framework based on *assume-guarantee reasoning*. More concretely, suppose we have a complex system $M_1 \parallel M_2$ composed of simpler pieces M_1 and M_2 , and we aim to check that a property P holds: $M_1 \parallel M_2 \models P$. Rather than checking P directly, we can apply the following rule:

$$\frac{\langle Q \rangle M_1 \langle P \rangle \quad \langle \text{true} \rangle M_2 \langle Q \rangle}{M_1 \parallel M_2 \models P} \quad (\text{R1})$$

This rule says that we can prove P by finding an *assumption* Q that (1) holds of M_2 and (2) implies that P holds of M_1 when Q holds of the rest of the system. Though it can be hard to craft such a Q , prior work [25, 51] shows that it can be inferred using L^* [6], a learning algorithm for finite automata, even for systems with multiple components. However, cryptographic protocols and their attacker models are usually expressed in specialized formalisms based on process calculi [1, 14] rather than automata. To bridge this gap, we developed a domain-specific language for modeling and verifying protocols, supporting a variety of constructs, such as symmetric and asymmetric encryption and digital signatures. The Taglierino Compiler compiles this language to finite automata (including one representing the attacker) that can be analyzed with L^* learning; our current implementation uses an extension of the LTS model checker [42, 51].

Taglierino uses *symbolic* cryptography [29], where the network is controlled by an attacker who can eavesdrop communication, replay and shuffle messages, but cannot sign a message without the corresponding private key or violate other basic cryptographic rules. Unlike other verifiers for this model [14, 53], Taglierino uses a bounded attacker model, and can only attest the absence of attacks up to a user-specified bound; if an attack exists, it can be found eventually by making this bound larger. Because of these limitations, we see Taglierino’s compositional analysis as complementary to existing monolithic, automated verification tools.

We demonstrate Taglierino by verifying several protocols from the literature. We focus on *authentication properties* [41, 56], but our approach could be adapted to other trace properties (e.g., weak or syntactic secrecy). Our case studies show that the generated assumptions can inform the design of protocol variants, whose components can be verified more efficiently and independently.

In summary, our *contributions* are:

- *Taglierino*, a compositional, automated protocol-verification framework based on assume-guarantee reasoning and L^* assumption learning.
- *Taglierino Compiler*, a Haskell library that automatically generates automata for protocol components, including a bounded symbolic attacker.
- The analysis of several *case-studies*, such as Needham-Schroeder-Lowe, Woo-Lam, and Kerberos, demonstrating that the tool can generate small, interpretable assumptions and speed up the verification of protocol variants.

This work extends an earlier paper reporting preliminary results [57] with: (1) an in-depth description of the Taglierino language and implementation, (2) a correctness proof of our verification approach, and (3) three new case studies.

2 Background

We briefly review labeled transition systems, which give semantics to protocols; and assumption learning and alphabet refinement, which Taglierino builds on.

2.1 Labeled Transition Systems

Messages in Taglierino are described by the grammar below, where n , k and $\{m\}k$ denote nonces, keys (symmetric or not), and an encrypted message.

$$\text{Term/Messages} \ni m ::= n \mid k \mid (m_1, \dots, m_n) \mid \{m\}k \mid \dots$$

Let $Act = \{send_i.m, recv_i.m, \dots\}$ be the set of *observable actions* (Sect. 3). A *labeled transition system* (LTS) is a tuple $M = (Q, \alpha M, \delta, q_0, Q_f)$, where

- Q is a set of *states*;
- $\alpha M \subseteq Act$, the *alphabet* of M , is a set of actions;
- $\delta \subseteq Q \times \Sigma \times Q$ is a *transition relation*;
- $q_0 \in Q$ is the initial state; and
- $Q_f \subseteq Q$ is the set of *accepting states*.

We say that M is *finite* if Q and αM are finite. The *language* of M , denoted $L(M)$, is the set of finite sequences $\sigma \in Act^*$ such that $q_0 \xrightarrow{\sigma} q_f$ for some $q_f \in Q_f$, where the \rightarrow relation is defined inductively as follows:

$$\frac{}{q \xrightarrow{\epsilon} q} \qquad \frac{(q, a, q') \in \delta \quad q' \xrightarrow{\sigma} q''}{q \xrightarrow{a \cdot \sigma} q''} \qquad \frac{a \notin \alpha M \quad q \xrightarrow{\sigma} q'}{q \xrightarrow{a \cdot \sigma} q'}$$

Here, ϵ is the empty sequence, and $a \cdot \sigma$ is the sequence that appends the action a to σ . The alphabet αM enumerates the actions that M controls; if $a \notin \alpha M$, a can occur without affecting M 's state. Given two LTSs M_1 and M_2 , their *parallel composition* $M_1 \parallel M_2$ synchronizes shared actions and interleaves the remaining actions. We use LTSs P to represent trace properties: M satisfies P if $L(M) \subseteq L(P)$.

2.2 Assumption Learning and Alphabet Refinement

The heart of our approach lies in L^* learning [6, 51], used by LTSA in the last stage of Taglierino. The algorithm looks for an assumption Q such that the assume-guarantee rule (R1) applies, where $\langle Q \rangle M \langle P \rangle$ is defined as $L(Q \parallel M) \subseteq L(P)$. If such a Q exists, it returns the weakest one that holds of M_2 . Otherwise, it outputs a counterexample that can be produced by $M_1 \parallel M_2$ but violates P .

In many cases, this procedure turns out to be too crude to be useful: if one of the components is too complex, Q will be expensive to compute and difficult to interpret. (In our case, the culprit is attacker model; cf. Sect. 5.) However, we can adapt the procedure to look for assumptions Q such that αQ is bounded by some finite alphabet $\Sigma \subseteq Act$. In practice, this leads to simpler assumptions that are faster to process and more interpretable. (Note, however, that they will generally be stronger than those found when learning is unconstrained.)

To find such a Σ automatically, we employ *alphabet refinement* [51]. The idea is to start with an empty alphabet and progressively add more labels until we find an assumption Q . A bit more formally, let M_1 and M_2 be the two system components, P be the property we want to check and $\Sigma_I \triangleq (\alpha M_1 \cup \alpha P) \cap \alpha M_2$ be the interface alphabet between M_1 and M_2 . The algorithm proceeds as follows:

1. Initialize $\Sigma := \emptyset$.
2. Run classic learning on Σ . If it finds some assumption Q , we stop. Otherwise, it returns some potential counterexample σ for $M_1 \parallel M_2 \models P$.
3. Find an index i such that $\sigma_i \in \Sigma_I$ but $\sigma_i \notin \Sigma$. If no i exists, σ is a real counterexample and we stop. Otherwise, set $\Sigma := \Sigma \cup \{\sigma_i\}$ and go to (2).

This procedure is guaranteed to terminate with either an assumption or a counterexample. To use it, we must choose some method for extracting an alphabet extension σ_i . In this paper, we always pick the smallest possible i .

3 An Overview of Taglierino

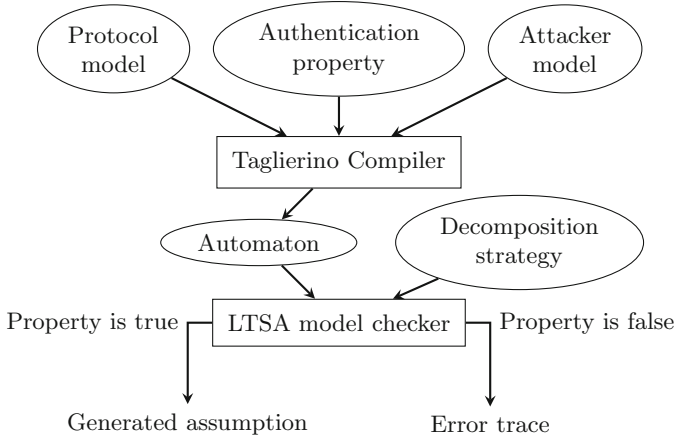
Taglierino takes as input a Haskell file with a protocol model, a set of specifications, and a description of the attacker. It compiles this input to LTSs for the LTSA model checker [42] extended with L^* learning [6, 51] (cf. Fig. 1). To verify the system, we decompose the system in LTSA, which generates assumptions for each component. If each component satisfies its assumptions, the protocol is correct. To illustrate, consider the following protocol:

Message 1. $A \rightarrow B : n_A$ Message 2. $B \rightarrow A : \{n_A\}sk_B$

Alice (A) generates a nonce n_A and sends it to Bob (B). Bob acknowledges by signing n_A with his private key sk_B and sending it back. By checking the signature, Alice knows that Bob did receive n_A and accepted her connection.

Figure 2 shows a model of this protocol. A (hidden) preamble declares constants such as `na` and the signing key `bobSK`. Each agent is defined using an embedded domain-specific language inspired by the applied pi calculus [1, 14]. Agents manipulate messages with cryptographic primitives (`sign`, `checkSig`, etc.), and communicate with `send` and `receive`. Their beliefs are expressed by `begin` and `end`: `begin` means that Bob is willing to establish a session keyed by `na`, whereas `end` means that Alice believes her connection attempt succeeded.

Our goal is to verify that the protocol satisfies agreement [41, 56]: if Alice thinks she’s talking to Bob, then Bob indeed accepted her connection—i.e., every

**Fig. 1.** Taglierino workflow

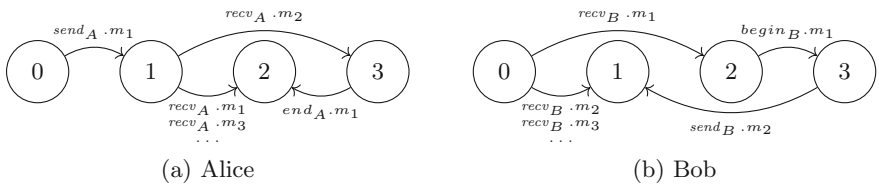
```

agent "Alice" (do
  send na
  sig  <- receive
  check <- checkSig bobPK sig na
  when check (end "auth" na))

agent "Bob" (do
  na <- receive
  begin "auth" na
  send (sign bobSK na))
  
```

Fig. 2. Simple authentication protocol in Taglierino

`end` is preceded by a matching `begin`. To do so, Taglierino compiles the agents to the LTSs in Fig. 3. The agents start at 0, and the transition labels represent interactions with the environment, including *begin* and *end* events. The indices represent the agent performing the action. For example, Alice first sends n_A and transitions to 1. There are multiple transitions from 1, depending on what she gets from the network. If she gets $\{n_A\}sk_B$, she moves to 3, emits $end_A.n_A$, and stops at 2; if she receives any other message, she skips $end_A.n_A$ and stops at 2.

**Fig. 3.** Honest agents, where $m_1 = n_A$, $m_2 = \{n_A\}sk_B$, and $m_3 = \{n_A\}sk_M$

Taglierino also produces an automaton that recognizes the traces allowed by the agreement property (Fig. 4). The automaton does not mention any *send*

or *receive* events, since those have no effect on agreement. *end* is allowed to occur multiple times, which corresponds to *weak*, or *non-injective* agreement [41]: a nonce can authenticate multiple sessions. Taglierino also supports *injective* agreement assertions [41], which forces each session key to be used at most once.

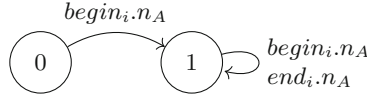


Fig. 4. Automaton for agreement assertion

For the agents to communicate, we need a network. Taglierino represents the network as another automaton, shown in Fig. 5. The labels are similar to those for agents, but the meanings of “send” and “receive” are swapped; e.g. $send_i.m$ means the network received m from the agent $i \in \{A, B\}$ (i.e., i sent m). The behavior of the entire system (Alice, Bob and the network) is given by the *parallel composition* of the LTSs of Figs. 3 and 5 (cf. Sect. 2).

The network automaton is much more complex than the others! In the symbolic model, the network abstracts all possible attacker behaviors, and its LTS describes all messages that an attacker can send after observing the agents’ actions. For example, the attacker can’t send $m_3 = \{n_A\}sk_B$ initially because it does not know n_A or the key sk_B . Still, the network LTS is only an approximation of the real symbolic model, which would require an infinite state space. The user can control this approximation through a parameter, the *allowed set*, which lists the messages that can be sent on the network; any other message is silently discarded. Here, only three messages are allowed: $m_1 = n_A$, $m_2 = \{n_A\}sk_B$, and $m_3 = \{n_A\}sk_M$, where sk_M is another signing key owned by the attacker.

Verifying the Protocol. When Alice receives Bob’s signature, she knows her connection was accepted because there is no other way such a message could have appeared: only Bob has the power to sign messages with sk_B , and since n_A hadn’t appeared in the network before Alice’s first message, the signature must have

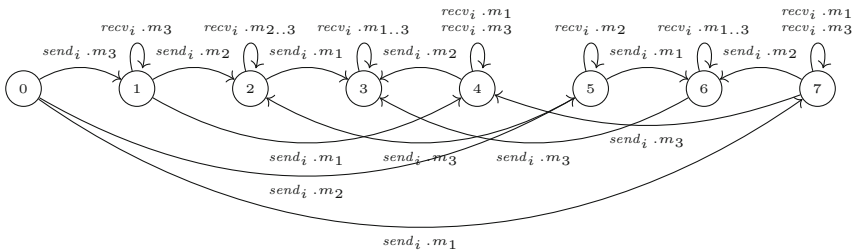


Fig. 5. Automaton for network ($m_1 = n_A$, $m_2 = \{n_A\}sk_B$, $m_3 = \{n_A\}sk_M$)

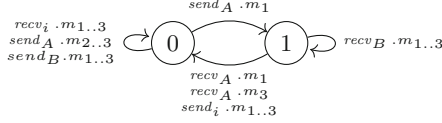


Fig. 6. Assumption Q for the component $M_2 = Net$

been created as a response to her. LTSA can verify this argument directly, but let's consider how to tackle this compositionally. To use the assume-guarantee rule (R1), we need to find a decomposition $S = M_1 \parallel M_2$ and an assumption Q for M_2 . We choose $M_1 = A \parallel B$ and $M_2 = Net$, and L^* infers the assumption Q in Fig. 6. The properties of L^* show the existence of Q is enough for ensuring the correctness of the protocol, but not only that. As we'll see (Sect. 5), we can reuse the assumptions to verify protocol variants, look for bugs, and sometimes even develop intuition for the protocol, informing potential refinements thereof.

The Taglierino API. (Figure 7) is structured around three types: **System**, **Term** and **Proc**. The first one is used for global declarations and compiler directives. The second one corresponds to the terms of Sect. 2.1, which are manipulated with symbolic primitives (`send`, `sdec`, etc.). The last one corresponds to processes. Processes can send terms to the network, receive them, and store them in a lookup table (cf. `insertFresh` and `store`). The `agent` function binds a process definition to an agent name. Each name can be bound to multiple processes, to model a protocol running multiple sessions.

Proc and **System** are monads [47], yielding a convenient syntax for models. For example, the `sig <- receive` notation of Fig. 2 binds `sig` to the received term, or, more generally, to the value returned by the right-hand side. Internally, **Proc** builds LTSs incrementally, in continuation-passing style. A value of type **Proc a** is roughly a function of type $(a \rightarrow \text{Automaton}) \rightarrow \text{Automaton}$, whose first argument is a callback that builds a partial LTS given a value $x :: a$. This simplifies the representation of network non-determinism. E.g., to implement `receive`, we use the callback `k :: Term -> Automaton` to compose the LTSs `k m`, where `m` ranges over all messages in the model's allowed set (cf. Sect. 4).

4 The Attacker Model and Its Correctness

A sent message m is learned by the attacker, who is free to manipulate it according to the rules of symbolic cryptography. The predicate $knows(K, m)$ (Fig. 8) says that the attacker can produce m after it has seen all messages in K . Roughly, the attacker may copy messages, extract components of a tuple, and encrypt, decrypt or sign messages with known keys. The $knows$ predicate yields two network LTSs. The first one is infinite, and describes the capabilities of an ideal attacker. The second one is a finite approximation of the first used in Taglierino.

Function name & Type	Description
<code>send :: Term -> Proc ()</code>	Send a message
<code>receive :: Proc Term</code>	Receive a message
<code>sign :: Term -> Term -> Term</code>	Use a key <code>k</code> to sign <code>m</code>
<code>checkSig :: Term -> Term -> Term -> Proc Bool</code>	Use <code>k</code> to check the sig. <code>s</code> with <code>m</code>
<code>aenc, senc :: Term -> Term -> Term</code>	Use a key <code>k</code> to encrypt <code>m</code>
<code>adec, sdec :: Term -> Term -> Proc Term</code>	Use a key <code>k</code> to decrypt <code>m</code>
<code>begin, end :: String -> Term -> Proc ()</code>	Agreement events
<code>agent :: Term -> Proc () -> System ()</code>	Agent definition
<code>gen* :: System Term</code>	Declare fresh nonces, keys, etc.
<code>allow :: [Term] -> System ()</code>	Add messages to allowed set
<code>knowledge Size :: Int -> System ()</code>	Bound attacker knowledge
<code>public :: [Term] -> System ()</code>	Declare public terms
<code>query :: String -> System ()</code>	Declare agreement assertion
<code>insertFresh :: Term -> Proc Bool</code>	Store <code>m</code> check if it is new
<code>store :: String -> [Term] -> System ()</code>	Declare private storage

Fig. 7. The Taglierino API

$$\begin{array}{c}
\frac{m \in K}{\text{knows}(K, m)} \quad \frac{\forall i \in \{1, \dots, n\}. \text{knows}(K, m_i)}{\text{knows}(K, f(m_1, \dots, m_n))} \quad \frac{\text{knows}(K, (m_1, \dots, m_n))}{\text{knows}(K, m_i)} \\
\\
\frac{\text{knows}(K, k) \quad \text{knows}(K, \{m\}k)}{\text{knows}(K, m)}
\end{array}$$

Fig. 8. Attacker knowledge; f ranges over operations of the symbolic model

Definition 1 (Network Automata). We define Net_K , the infinite network automaton, for a finite set of terms K (the initial knowledge). Its states are all finite sets of terms; the initial state is \emptyset and all states are accepting. The alphabet consists of all $send$ and $recv$ events. The transitions in Net_K are (1) $K' \xrightarrow{send_i.m} K' \cup \{m\}$, and (2) $K' \xrightarrow{recv_i.m} K'$ when $\text{knows}(K \cup K', m)$.

Definition 2 (Finite Network Automata). We define $Net_{K,A,k}$, the bounded network automaton, for K , a (finite) allowed set $A \supseteq K$, and a bound $k \in \mathbb{N}$. Its states are all subsets of A of size at most k ; the initial state is \emptyset and all states are accepting. Its transitions are (1) $K' \xrightarrow{send_i.m} K'$, (2) $K' \xrightarrow{send_i.m} K' \cup \{m\}$ when $|K'| < k$, and (3) $K' \xrightarrow{recv_i.m} K'$ when $\text{knows}(K \cup K', m)$.

The states represent the messages that the attacker knows beyond the initial knowledge K . The $send$ transitions allow the attacker to incorporate a message into its knowledge—in the bounded case, only if we do not exceed the bound k . The $recv$ transitions allow the attacker to deliver any messages in the knowledge. Since any behavior of Net_K can be observed in $Net_{K,A,k}$ for A and k large enough, any symbolic attack can be caught by Taglierino.

Theorem 1. For all K , $L(Net_K) = \bigcup_{A,k} L(Net_{K,A,k})$.

Corollary 1 (Soundness and Relative Completeness). Let M and P be finite. Then $L(M \parallel Net_K) \subseteq L(P) \iff \forall A, k, L(M \parallel Net_{K,A,k}) \subseteq L(P)$.

Here, M corresponds to the parallel composition of all honest agents, and P corresponds to the property to be verified. Strictly speaking, the bound k is not needed to ensure finiteness. However, if $|A| = n$, we need 2^n states in the worst case to represent an attacker. To make verification tractable, we would need to keep n small, preventing us from exploring interesting attacker behavior. With k , we just need $\sum_{i=0}^k \binom{n}{i}$ states to represent the attacker, which grows polynomially in k for fixed n . We don't need k to verify the example of Sect. 3, but this parameter will be crucial for tackling more complex protocols; cf. Sect. 5.

5 Protocol Analysis

We evaluated Taglierino by verifying a series of protocols from the literature. We were interested in the following research questions:

- RQ1. Is compositional verification more efficient than monolithic verification?
- RQ2. Can assume-guarantee reasoning help the verification of protocol variants?
- RQ3. Can the learned assumptions provide insight into the design of a protocol?

Our results show that compositional verification is usually cheaper than monolithic verification. Though expensive to learn, the assumptions can be reused on protocol variants, often reducing the verification time by 2–5×. Moreover, they are often interpretable, highlighting which protocol messages are important.

We proceed as follows. We present our setup (Sect. 5.1) and overview each case study: Needham-Schroeder-Lowe [40, 49] (Sect. 5.2), Denning-Sacco [28] (Sect. 5.3), Woo-Lam [55] (Sect. 5.4) and Kerberos [38] (Sect. 5.5). We discuss the implications of each case study for RQ2 and RQ3. Whenever meaningful, we'll interpret the learned assumptions. We conclude by summarizing our results (Sect. 5.6), answering RQ1 with quantitative and performance data.

5.1 Evaluation Setup

We encoded the protocols with the API of Fig. 7. The model includes nonces and keys for honest agents and for the corrupt Mallory (M). Only M 's data is part of the initial attacker knowledge. Recall that, to generate the attacker, we must declare which messages can go in the network, and how many M can remember (Sect. 4). We chose the allowed set by starting with all messages exchanged in a good run, and then adding other messages by mutating some of the parameters of the first group (e.g. replacing an honest nonce with a nonce generated by M). This set ranged from a few messages to 170. We allowed 2 to 4 additional messages in M 's knowledge, since larger bounds caused the model to explode.

This was enough to explore interesting behaviors and to observe known bugs from the literature. Indeed, these bugs do not hinge on a large knowledge, but on few critical pieces of information, such as a particular nonce or signature.

After compiling the models, we analyzed them in LTSA. We’ve seen that the network is much larger than the other generated LTSs (Sect. 3), and in our case studies it easily reached hundreds of thousands of states. Thus, to obtain useful assumptions and a tractable analysis, we employed two decomposition steps. Suppose that the protocol comprises Alice (A), Bob (B) and the network. First, we generated an assumption Q_{Net} for the network using alphabet refinement (Sect. 2.2), letting $M_2 = Net$ and $M_1 = A || B$ in (R1). (As our version of LTSA does not implement this algorithm, we ran it manually, using the tool to generate assumptions for each candidate alphabet.) By replacing the attacker with Q_{Net} , the protocol became much easier to analyze. We then learned an assumption Q_A for A by setting $M_2 = A$ and $M_1 = B || Q_{Net}$. Finally, we reused Q_A to verify alternative implementations A' for Alice. If A' satisfies Q_A , the variant is secure against the same bounded attacker. Otherwise, we obtain a counterexample showing that A' does not satisfy Q_A . In principle, this counterexample might be spurious, since Q_A is generated using $B || Q_{Net}$, which includes more behaviors than $B || Net$. To rule out this possibility, we check that the counterexample can be produced by $A' || B || Net$. All experiments were performed on a 1.6 GHz Intel Core i5 CPU and 8.0 GB RAM, running 64-bit Ubuntu 18.04 LTS.

5.2 Needham-Schroeder-Lowe

The Needham-Schroeder-Lowe (NSL) protocol [40, 49] attempts to provide mutual authentication between two parties:

- | | |
|---|---|
| (1) $A \rightarrow S : A, B$ | (5) $S \rightarrow B : \{A, pk_A\}sk_S$ |
| (2) $S \rightarrow A : \{B, pk_B\}sk_S$ | (6) $B \rightarrow A : \{n_A, n_B, B\}pk_A$ |
| (3) $A \rightarrow B : \{n_A, A\}pk_B$ | (7) $A \rightarrow B : \{n_B\}pk_B$ |
| (4) $B \rightarrow S : B, A$ | |

Alice (A) starts by asking a key server S for Bob’s public key pk_B . (In our model, the peer is chosen by the attacker.) The server replies to A signing the reply with its own secret key sk_S . Then, A encrypts a fresh n_A and sends it to B , along with her identity. Bob asks S for A ’s public key pk_A , and then sends n_A back to her along another fresh nonce n_B and his identity, all of this encrypted with pk_A . Finally, A acknowledges the end of the handshake to B by sending n_B back.

Informally, we want to show: when A receives (6), she knows that B accepted her connection; and when B receives (7), he knows that A tried to contact him. We analyzed the second property, as it allows us to explore a broken variant (Sect. 5.2); the first one is similar and thus omitted. We allowed 31 messages and bounded M ’s knowledge to 4 messages. We generated an assumption for Net using alphabet refinement, and then one for A . Figure 9 shows the alphabets.

Protocol Variant: NS. The NSL protocol fixed a vulnerability in the earlier NS protocol [40, 49], which resulted from omitting B in (6). We simulated NS by

$send_i(\{n_A, n_B, M\}pk_A)$	$recv_i(\{n_A, n_B, M\}pk_A)$	$send_i(\{n_B\}pk_B)$
$send_i(\{n_A, n_B, B\}pk_M)$	$recv_i(\{n_A, n_B, B\}pk_M)$	$send_i(\{n_B\}pk_M)$
$send_i(\{n_A, n_B, M\}pk_M)$	$recv_i(\{n_A, n_B, M\}pk_M)$	$send_i(\{B, pk_B\}sk_S)$
$send_i(\{n_B, n_B, B\}pk_M)$	$recv_i(\{n_B, n_B, B\}pk_M)$	$recv_i(\{n_B\}pk_B)$
$send_i(\{n_B, n_B, M\}pk_M)$	$recv_i(\{n_B, n_B, M\}pk_M)$	$recv_i(\{n_B\}pk_M)$
$send_i(\{n_M, n_B, B\}pk_M)$	$recv_i(\{n_M, n_B, B\}pk_M)$	$recv_i(\{B, pk_B\}sk_S)$
$send_i(\{n_M, n_B, M\}pk_M)$	$recv_i(\{n_M, n_B, M\}pk_M)$	$begin_A(auth_{AB}, B)$
		$begin_A(auth_{AB}, M)$

Fig. 9. NSL assumption alphabets for Alice and Mallory. The identifier i ranges over A, B and S for Mallory, and over A for Alice. Only Alice uses *begin*.

simply not checking that identity when A receives (6). Then, A ends up breaking its assumption, and the counterexample reveals the original attack:

(1) $M \rightarrow A : M$	(7) $M \rightarrow B : \{n_A, A\}pk_B$
$begin_A(auth_{AB}, M)$	(8) $B \rightarrow S : B, A$
(2) $A \rightarrow S : A, M$	(9) $S \rightarrow B : \{A, pk_A\}sk_S$
(3) $S \rightarrow A : \{M, pk_M\}sk_S$	(10) $B \rightarrow A : \{n_A, n_B, B\}pk_A$
(4) $A \rightarrow M : \{n_A, A\}pk_M$	(11) $A \rightarrow M : \{n_B\}pk_M$
(5) $M \rightarrow S : A, B$	(12) $A \rightarrow B : \{n_B\}pk_B$
(6) $S \rightarrow M : \{B, pk_B\}sk_S$	$end_B(auth_{AB}, B)$

Variants: Serverless NSL. A common simplification is to assume that A already knows the keys of her peer, obviating the need for (1) and (2). LTSA reports that this modification also satisfies A 's assumptions, implying security.

Interpreting the Assumptions. We abstract M and A 's behavior using assumption learning with alphabet refinement. The alphabets (Fig. 9) list the actions that must be controlled for the property to hold; removing them means allowing the M to freely perform them, regardless of whether a send action was triggered by an honest agent or of whether M had enough knowledge to deliver a message. In other words, if an action is in M 's alphabet, it must be synchronized with her automaton; otherwise, it can be directly triggered by the agents.

The difference between M and A 's alphabets is that M 's includes actions for B , whereas A 's includes her own actions and the *begin* events. Most of the controlled actions are variants of (6) encrypted with pk_M . If M can forge such messages, she can learn n_B even before B is contacted, thus obtaining the information needed to impersonate A and break agreement. (Note that we didn't include n_B in the allowed set of messages, so it is not possible for M to learn this value directly.) Interestingly, the expected message (6) in a normal run of the protocol, $\{n_A, n_B, B\}pk_A$, is not in the alphabet. Intuitively, since M does not have pk_A , the only thing she can do with this message is relaying it to A . If A meant to talk to B anyway, A will eventually trigger *begin* and send her response (7), which does not pose any harm for agreement. Otherwise, if A meant to talk to M , B 's identity will not match M 's, and A stops without sending (7) to B .

5.3 Denning-Sacco

The Denning-Sacco protocol [28] is used to agree on a shared symmetric key given by a trusted server. We consider the following simplified version without timestamps (where $CA = \{A, pk_A\}sk_S$ and $CB = \{B, pk_B\}sk_S$):

- (1) $A \rightarrow S : A, B$ (3) $A \rightarrow B : CA, CB, \{\{A, B, k_{AB}\}sk_A\}pk_B$
 (2) $S \rightarrow A : CA, CB$

This version uses a defense [2] against an attack similar to that of Sect. 5.2. The specification is that (3) should prove to B that A has tried to contact him.

Our model allows 170 messages in the network and 2 additional messages in the attacker knowledge. After generating an attacker assumption with alphabet refinement, we used it to generate an assumption for B .

A Broken Variant. When B receives (3), he knows that A wants to start a connection and the key is shared between the two. We modified B so that the identity in (3) is not checked, which is equivalent to the original broken version. Thus, B accepts $CA, CB, \{\{A, M, k_{AM}\}sk_A\}pk_B$ and believes he is contacting A . This behavior enables the attack on Denning-Sacco [2], which we rediscovered by checking the modified B against his assumption.

We did not consider a serverless variant of Denning-Sacco like we did earlier. In practice, certificates issued by S also contain timestamps to prevent replays. Thus, it wouldn't make sense to assume Alice knows the certificate from the start nor to assume Alice tries to contact the agents she wants using old certificates.

$$\begin{array}{l}
 \alpha_i(CA, CB) \\
 \alpha_i(CA, CB, \{\{A, B, k_{AB}\}sk_A\}pk_B) \quad \alpha_i(CA, CB, \{\{A, B, k_{AM}\}sk_A\}pk_B) \\
 \alpha_i(CA, CB, \{\{A, M, k_{AB}\}sk_A\}pk_B) \quad \alpha_i(CA, CB, \{\{A, M, k_{AM}\}sk_A\}pk_B) \\
 \alpha_i(CA, CB, \{\{B, A, k_{AB}\}sk_A\}pk_B) \quad \alpha_i(CA, CM, \{\{A, B, k_{AB}\}sk_A\}pk_B) \\
 \alpha_i(CB, CA, \{\{A, B, k_{AB}\}sk_A\}pk_B) \quad \alpha_i(CB, CM, \{\{A, B, k_{AB}\}sk_A\}pk_B) \\
 \alpha_i(CM, CA, \{\{A, B, k_{AB}\}sk_A\}pk_B) \quad \alpha_i(CM, CB, \{\{A, B, k_{AB}\}sk_A\}pk_B) \\
 \alpha_i(CA, CB, \{\{A, B, k_{AB}\}sk_A\}pk_M) \quad \alpha_i(CA, CM, \{\{A, B, k_{AB}\}sk_A\}pk_M) \\
 \alpha_i(CB, CA, \{\{A, B, k_{AB}\}sk_A\}pk_M) \quad \alpha_i(CB, CM, \{\{A, B, k_{AB}\}sk_A\}pk_M) \\
 \alpha_i(CM, CA, \{\{A, B, k_{AB}\}sk_A\}pk_M) \quad \alpha_i(CM, CB, \{\{A, B, k_{AB}\}sk_A\}pk_M)
 \end{array}$$

Fig. 10. Refined attacker alphabet in Denning-Sacco; $i \in \{A, B, S\}$, $\alpha \in \{send, recv\}$, $CA = \{A, pk_A\}sk_S$, $CB = \{B, pk_B\}sk_S$, $CM = \{M, pk_M\}sk_S$.

Interpreting the Assumptions. Figure 10 shows the assumption alphabet. Most actions are variants of (3). The first two components are not important since they are public. The last component is encrypted with either pk_B or pk_M and contains Alice's signature on the shared keys. If the attacker is free to forge such messages, he can fake (3) even when Alice is offline and break security.

$$\alpha_i(n_B), \alpha_i(\{B, n_B\}k_{AS}), \alpha_i(\{A, n_B\}k_{BS}), \alpha_i(A, B, \{B, n_B\}k_{AS}), \alpha_i(M, B, \{B, n_B\}k_{AS})$$

Fig. 11. Refined attacker alphabet in Woo-Lam; $i \in \{A, B, S\}$, $\alpha \in \{send, recv\}$.

5.4 Woo-Lam

Woo and Lam [55] present a symmetric-key protocol two agents authenticate via a trusted server. We use a modified version [36], which includes two fixes [2, 4] that were not present originally:

- | | |
|--|--|
| (1) $A \rightarrow B : A$
(2) $B \rightarrow A : n_B$
(3) $A \rightarrow B : \{B, n_B\}k_{AS}$ | (4) $B \rightarrow S : A, B, \{B, n_B\}k_{AS}$
(5) $S \rightarrow B : \{A, n_B\}k_{BS}$ |
|--|--|

The goal of the protocol is to ensure that, when B receives (5), he knows that A is online and has responded to n_B . Our model allows 20 messages in the network and allows the attacker to learn at most 2 messages in addition to her initial knowledge. We modeled two instances of B , but checked the agreement for only one of them, since LTSA was taking too long to generate assumptions for the two. Again, we used two decomposition steps.

A Broken Variant. We modified B to reintroduce a flaw found in the original protocol [2], by removing the check on the identity in (5). We rediscovered the flaw by checking the modification against his generated assumption.

Interpreting the Assumptions. Figure 11 shows the refined alphabet of the attacker in the fixed protocol. Unlike in Sect 5.2, (5) *does* appear in the refined alphabet. In NSL, the authentication of A hinges on the secrecy of n_B , which is protected by pk_A , whereas here n_B is public and the crucial factor is the secrecy of the shared keys between the agents and the server. Forging (5) means effectively learning this key and breaking agreement.

5.5 Kerberos

Kerberos [38] is a protocol designed to provide mutual authentication between two parties via a trusted server S . We model a common simplification [2, 20, 50]:

- | | |
|---|---------------------------------------|
| (1) $A \rightarrow S : A, B, n_A$
(2) $S \rightarrow A : \{k_{AB}, A\}k_{BS}, \{k_{AB}, B, n_A\}k_{AS}$
(3) $A \rightarrow B : \{k_{AB}, A\}k_{BS}, \{A, t_A\}k_{AB}$ | (4) $B \rightarrow A : \{t_A\}k_{AB}$ |
|---|---------------------------------------|

Upon receiving n_A , S generates a new key k_{AB} and creates a ticket for B ($\{k_{AB}, A\}k_{BS}$) secured using k_{BS} , as well as a response for A . She decrypts her part to learn k_{AB} , while checking that the identifier and the nonce match those sent initially. She then constructs an authenticator containing a timestamp

Protocol	Component	Original		Assumption	
		#States	#Trans.	#States	#Trans.
NSL public key	Attacker	775030	4343487	3	178
	Alice	14	163	6	69
Denning-Sacco	Attacker	719999	3711528	2	203
	Bob	3	171	3	103
Woo-Lam	Attacker	4374	28016	13	516
	Bob	3844	75888	8	161
Kerberos	Attacker	13528	85840	4	252
	Bob	26	228	5	126

Fig. 12. Sizes of components and assumptions, in terms of states and transitions.

t_A encrypted with the session key k_{AB} . She sends the ticket and the authenticator to B , who then learns k_{AB} and t_A . He checks that the ticket and the authenticator match before ending the handshake.

The goals of the protocol are as follows: when B receives (3), he knows that A wants to start a session with him; when A receives (4), she knows that B accepts her connection. Unfortunately [11], Mallory can replay (3) and trick B into thinking that A is attempting to setup two or more simultaneous sessions with him, when in reality Alice is trying to establish only one session. The fix is for B to store all live authenticators in order to detect the replay, though this might be infeasible for a practical implementation. To demonstrate this fix, we introduced a session lookup table in B (cf. Fig. 7). This allows rejecting a replay of (3) when the authenticator was stored in another session. We created two parallel instances of B in the model. We checked authentication of A for detecting flaws in the broken variants of the protocol. The other property, authentication of B , is similar. In total, our model allows 22 messages in the network and allows 2 additional messages in the initial knowledge. As usual, we performed two decomposition steps.

A Broken Variant. We modified B in the fixed version so that the received authenticators are not checked for freshness. This behavior enables the above attack, which we rediscovered by checking the new B against the assumption.

Interpreting the Assumptions. We inspected B 's assumption. We observed that, when emitting *end* after (3), he transitions to a loop where his two instances can only *send* or *recv* but cannot trigger another *end*. Intuitively, since we only model one instance of A , she will only send one authenticator to B . Thus, once he receives the authenticator, further receives are replays and should be ignored.

5.6 Performance Evaluation Results

We evaluate the effect of alphabet refinement, and then compare the effort required by compositional and monolithic verification (RQ1). Figure 12 shows that alphabet refinement abstracts the behavior of the attacker and agents and

significantly reduces their complexity: the abstractions (“Assumption”) are much smaller than the original automata (“Original”). Figure 13 compares monolithic and compositional verification in terms of the size of the automata and the time for checking their properties. The Attack column specifies whether the variant is vulnerable to an attack. The Bounds column shows the attacker bounds we used: the first number refers to the size of allowed set and the second, the number of messages attacker can learn in addition to its initial knowledge. The next column shows how long it took to compile the various automata produced by Taglierino. The next three columns show the resources used for monolithic verification and the last three columns show the resources used for compositional verification.

Rows marked with (*) are the variants used to generate the assumptions. In principle, we didn’t need to carry out a separate compositional verification of those variants (since this is already done when generating the assumptions), but we repeated the verification for completeness. Rows marked (**) are buggy variants that yielded counterexamples in LTSA. As explained earlier, we need to rule out false positives, so the performance figures include two numbers: the first is the time for generating the counterexample and the second is the time for validating it (in parentheses).

To illustrate, consider the first two rows of Fig. 13 which refer to checking Needham-Schroeder-Lowe (NSL) and Needham-Schroeder (NS) protocols in our framework. The column “Attack” shows that using our framework we did not find an attack in NSL and we did find an attack in NS. The column “Bounds” shows the bounds used in the models: 31 messages are allowed, and the attacker can learn at most 4 messages in addition to the initial knowledge. Under “Monolithic verification” we show the cost of verifying the protocols monolithically (i.e. checking $Alice \parallel Bob \parallel Server \parallel Network \models P$), whereas “Compositional verification” has the cost of verifying the protocol compositionally. The first row (NSL) shows the effort of checking $Alice \models Q_{Alice}$ where Q_{Alice} is the generated assumption for Alice. The numbers on the second row (NS) are the cost of checking $Alice_{NS} \models Q_{Alice}$ where $Alice_{NS}$ is the automaton for Alice. This check produced a counterexample $Alice'$, and the numbers in parentheses are the cost of confirming it (i.e. checking $Alice' \parallel Bob \parallel Server \models P$).

We observe that compositional verification requires substantially fewer resources than monolithic verification, with the only exception being the broken variant of the Woo-Lam protocol. However, these numbers do not include the time spent to generate Alice’s assumption, which amounts to approximately 1 to 5 min, implying that the benefits of compositional verification mostly apply when we expect to reuse the generated assumptions for several protocol variants.

Protocol	Attack	Bounds	Compile time	Monolithic verification			Compositional verification			
				#States	#Transitions	Time	#States	#Transitions	Time	
NSL [41]	No	31, 4	2851	388	2738	8	18	163	1	*
NS [51]	Yes [41]	31, 4	2674	12425	116365	97	19 (3547)	164 (25703)	1 (22)	**
NSL (w/o server)	No	31, 1	2182	11142	97517	115	13	99	1	
DS (fixed) [2]	No	170, 2	909	2308	14722	63	4	171	1	*
DS (broken) [29]	Yes [2]	170, 2	974	1913	13939	43	4 (956)	172 (3944)	1 (24)	**
WL (fixed) [37]	No	20, 2	1214	58189	441161	215	4078	78207	8	*
WL (broken) [57]	Yes [2]	20, 2	1209	7493	82593	22	2555 (58593)	46221 (358567)	6 (76)	**
KB (fixed)	No	22, 2	1295	434	1988	12	31	332	1	*
KB [39]	Yes [11]	22, 2	1029	420	1993	10	33 (90)	337 (200)	1 (1)	**

Fig. 13. Evaluation results. We use the following abbreviations: NSL for Needham-Schroeder-Lowe; NS for Needham-Schroeder; DS for Denning-Sacco; WL for Woo-Lam; KB for Kerberos. All time figures are in ms. Rows marked with (*) are the variants used to generate the assumptions. Rows marked (**) are buggy variants that yielded counterexamples. We report the time to produce a counterexample and the time to validate the counterexample (inside the parenthesis).

6 Related Work

Many researchers have investigated formal methods for security protocols [7, 10, 12, 14, 15, 18, 27, 30, 31, 34, 37, 39, 53, 54]. We discuss here research areas that are closely related to Taglierino: automated cryptographic protocol verification, and compositional reasoning.

Researchers have developed automated analysis tools for protocols, most of which are based on the symbolic model like Taglierino [17, 26, 32, 35, 45], while some uses the computational model [8, 9, 16, 33]. Unlike ours, these tools are monolithic, analyzing the entire protocol at once, and thus cannot easily reuse intermediary results to verify protocol variants like we do. Interestingly, Tamarin [53] allows users to specify intermediate lemmas to aid verification, but this still requires the entire protocol code. It would be interesting to investigate how to integrate the properties discovered by Taglierino in such a framework.

Compositional verification and assume-guarantee reasoning [43, 44, 46, 48, 52] have been studied extensively to address the state-space explosion problem [24]. It has been applied to protocol verification, notably in the Protocol Compositional Logic (PCL) [27]. PCL targets security properties of cryptographic protocols, supporting compositional reasoning about complex security protocols. It has been applied to a number of industry standards including SSL/TLS, IEEE 802.11 i and Kerberos V5. Despite its success, PCL is limited by the large manual effort that is required by the proofs. An automatic compositional approach for security analysis is presented in [5]; that work targets protocols composed from smaller protocols. It does not use assume-guarantee reasoning but instead uses a property of independence that enables analysis of component protocols in isolation. A quite different approach [13], proposes a modular code verification method for protocol implementations. The method is based on checking invariants on the usage of cryptography. Invariants are expressed as refinement types which can be checked efficiently using type checking.

In the model checking community, progress has been made in automating compositional reasoning using learning and abstraction-refinement techniques for iterative building of the necessary assumptions [19, 25, 51]. Other learning-based approaches for automating assumption generation have been proposed as well, e.g. [3, 21–23], with many other results to follow. As future work, we will investigate incorporating more advanced learning techniques in Taglierino.

7 Conclusion

We developed Taglierino to analyze authentication protocols, featuring a compiler for generating automata for agents in the protocol and a bounded Dolev-Yao attacker. Our framework allows us to synthesize assumptions for protocol components which can be used to compositionally verify protocol variants and to provide insights into the protocol design. Our results show that compositional verification yields faster checks when we the assumption can be reused.

As our compiler is quite general, we plan to investigate how to integrate it with more powerful model checking frameworks. Furthermore we plan to study learning and abstraction-based methods that would allow us to automate more involved compositional proofs, that use circular reasoning (necessary for instance to reason about secrecy) and put no bound on the search space.

References

1. Abadi, M., Blanchet, B., Fournet, C.: The applied pi calculus: mobile values, new names, and secure communication. CoRR abs/1609.03003 (2016). <http://arxiv.org/abs/1609.03003>
2. Abadi, M., Needham, R.: Prudent engineering practice for cryptographic protocols. *IEEE Trans. Softw. Eng.* **22**(1), 6–15 (1996). <https://doi.org/10.1109/32.481513>
3. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 548–562. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_52
4. Anderson, R., Needham, R.: Programming Satan’s computer. In: van Leeuwen, J. (ed.) *Computer Science Today*. LNCS, vol. 1000, pp. 426–440. Springer, Heidelberg (1995). <https://doi.org/10.1007/BFb0015258>
5. Andova, S., Cremers, C., Gjøsteen, K., Mauw, S., Mjølsnes, S.F., Radomirović, S.: A framework for compositional verification of security protocols. *Inf. Comput.* **206**(2), 425–459 (2008). *Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (FCS-ARSPA 2006)*. <https://doi.org/10.1016/j.ic.2007.07.002>. <http://www.sciencedirect.com/science/article/pii/S0890540107001228>
6. Angluin, D.: Learning regular sets from queries and counterexamples. *Inf. Comput.* **75**(2), 87–106 (1987). [https://doi.org/10.1016/0890-5401\(87\)90052-6](https://doi.org/10.1016/0890-5401(87)90052-6)
7. Barthe, G., Grégoire, B., Béguelin, S.Z.: Formal certification of code-based cryptographic proofs. In: Shao, Z., Pierce, B.C. (eds.) *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, 21–23 January 2009*, pp. 90–101. ACM (2009). <https://doi.org/10.1145/1480881.1480894>

8. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-aided security proofs for the working cryptographer. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 71–90. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22792-9_5
9. Barthe, G., Grégoire, B., Heraud, S., Zanella Béguelin, S.: Formal certification of ElGamal encryption. In: Degano, P., Guttman, J., Martinelli, F. (eds.) FAST 2008. LNCS, vol. 5491, pp. 1–19. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01465-9_1
10. Basin, D.A., Dreier, J., Hirschi, L., Radomirovic, S., Sasse, R., Stettler, V.: A formal analysis of 5G authentication. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, 15–19 October 2018, pp. 1383–1396. ACM (2018). <https://doi.org/10.1145/3243734.3243846>
11. Bellare, S.M., Merritt, M.: Limitations of the Kerberos authentication system. SIGCOMM Comput. Commun. Rev. **20**(5), 119–132 (1990). <https://doi.org/10.1145/381906.381946>
12. Bhargavan, K., Blanchet, B., Kobeissi, N.: Verified models and reference implementations for the TLS 1.3 standard candidate. In: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, 22–26 May 2017, pp. 483–502. IEEE Computer Society (2017). <https://doi.org/10.1109/SP.2017.26>
13. Bhargavan, K., Fournet, C., Gordon, A.D.: Modular verification of security protocol code by typing. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, 17–23 January 2010, pp. 445–456 (2010). <https://doi.org/10.1145/1706299.1706350>
14. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: 14th IEEE Computer Security Foundations Workshop (CSFW-14 2001), Cape Breton, Nova Scotia, Canada, 11–13 June 2001, pp. 82–96. IEEE Computer Society (2001). <https://doi.org/10.1109/CSFW.2001.930138>
15. Blanchet, B.: A computationally sound mechanized prover for security protocols. In: 2006 IEEE Symposium on Security and Privacy (S&P 2006), Berkeley, California, USA, 21–24 May 2006, pp. 140–154. IEEE Computer Society (2006). <https://doi.org/10.1109/SP.2006.1>
16. Blanchet, B.: A computationally sound mechanized prover for security protocols. In: Proceedings of the 2006 IEEE Symposium on Security and Privacy, SP 2006, pp. 140–154. IEEE Computer Society (2006). <https://doi.org/10.1109/SP.2006.1>
17. Blanchet, B.: Modeling and verifying security protocols with the applied Pi calculus and ProVerif. Found. Trends Priv. Secur. **1**(1–2), 1–135 (2016). <https://doi.org/10.1561/33000000004>
18. Blanchet, B.: Symbolic and computational mechanized verification of the ARINC823 avionic protocols. In: 30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, 21–25 August 2017, pp. 68–82. IEEE Computer Society (2017). <https://doi.org/10.1109/CSF.2017.7>
19. Gheorghiu Bobaru, M., Păsăreanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 135–148. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_14
20. Burrows, M., Abadi, M., Needham, R.: A logic of authentication. ACM Trans. Comput. Syst. **8**(1), 18–36 (1990). <https://doi.org/10.1145/77648.77649>

21. Chaki, S., Clarke, E., Sinha, N., Thati, P.: Automated assume-guarantee reasoning for simulation conformance. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 534–547. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_51
22. Chen, Y.-F., Clarke, E.M., Farzan, A., Tsai, M.-H., Tsay, Y.-K., Wang, B.-Y.: Automated assume-guarantee reasoning through implicit learning. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 511–526. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_44
23. Chen, Y.-F., Farzan, A., Clarke, E.M., Tsay, Y.-K., Wang, B.-Y.: Learning minimal separating DFA's for compositional verification. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 31–45. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_3
24. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
25. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_24
26. Cremers, C.J.F.: The Scyther Tool: Verification, falsification, and analysis of security protocols. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 414–418. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_38
27. Datta, A., Derek, A., Mitchell, J.C., Roy, A.: Protocol composition logic (PCL). *Electron. Notes Theor. Comput. Sci.* **172**, 311–358 (2007). <https://doi.org/10.1016/j.entcs.2007.02.012>
28. Denning, D.E., Sacco, G.M.: Timestamps in key distribution protocols. *Commun. ACM* **24**(8), 533–536 (1981). <https://doi.org/10.1145/358722.358740>
29. Dolev, D., Yao, A.: On the security of public key protocols. *IEEE Trans. Inf. Theory* **29**(2), 198–208 (1983)
30. Dreier, J., Hirschi, L., Radomirovic, S., Sasse, R.: Verification of stateful cryptographic protocols with exclusive OR. *J. Comput. Secur.* **28**(1), 1–34 (2020). <https://doi.org/10.3233/JCS-191358>
31. Erbsen, A., Philipoom, J., Gross, J., Sloan, R., Chlipala, A.: Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In: 2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, 19–23 May 2019, pp. 1202–1219. IEEE (2019). <https://doi.org/10.1109/SP.2019.00005>
32. Escobar, S., Meadows, C., Meseguer, J.: Maude-NPA: cryptographic protocol analysis modulo equational properties. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007-2009. LNCS, vol. 5705, pp. 1–50. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03829-7_1
33. Fournet, C., Kohlweiss, M., Strub, P.Y.: Modular code-based cryptographic verification. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, pp. 341–350. Association for Computing Machinery, New York (2011). <https://doi.org/10.1145/2046707.2046746>
34. Fromherz, A., Giannarakis, N., Hawblitzel, C., Parno, B., Rastogi, A., Swamy, N.: A verified, efficient embedding of a verifiable assembly language. *Proc. ACM Program. Lang.* **3**(POPL), 63:1–63:30 (2019). <https://doi.org/10.1145/3290376>
35. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3—a modern refinement checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 187–201. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_13

36. Gordon, A.D., Jeffrey, A.: Authenticity by typing for security protocols. In: 2001 Proceedings of the 14th IEEE Computer Security Foundations Workshop, pp. 145–159 (2001)
37. Kobeissi, N., Bhargavan, K., Blanchet, B.: Automated verification for secure messaging protocols and their implementations: a symbolic and computational approach. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, 26–28 April 2017, pp. 435–450. IEEE (2017). <https://doi.org/10.1109/EuroSP.2017.38>
38. Kohl, J., Neuman, C., et al.: The Kerberos network authentication service (V5). Technical report, RFC 1510, September 1993
39. Liao, K., Hammer, M.A., Miller, A.: ILC: a calculus for composable, computational cryptography. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, 22–26 June 2019, pp. 640–654. ACM (2019). <https://doi.org/10.1145/3314221.3314607>
40. Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 147–166. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61042-1_43
41. Lowe, G.: A hierarchy of authentication specifications. In: Proceedings 10th Computer Security Foundations Workshop, pp. 31–43. IEEE (1997)
42. Magee, J., Kramer, J.: Concurrency: State Models and Java Programs. Wiley, Hoboken (1999)
43. McMillan, K.L.: Verification of an implementation of Tomasulo’s algorithm by compositional model checking. In: Hu, A.J., Vardi, M.Y. (eds.) CAV 1998. LNCS, vol. 1427, pp. 110–121. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0028738>
44. McMillan, K.L.: Circular compositional reasoning about liveness. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 342–346. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48153-2_30
45. 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
46. Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE Trans. Softw. Eng. **7**(4), 417–426 (1981)
47. Moggi, E.: Computational lambda-calculus and monads. In: Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS 1989), Pacific Grove, California, USA, 5–8 June 1989, pp. 14–23. IEEE Computer Society (1989). <https://doi.org/10.1109/LICS.1989.39155>
48. Namjoshi, K.S., Treffer, R.J.: On the completeness of compositional reasoning. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 139–153. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_14
49. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM **21**(12), 993–999 (1978). <https://doi.org/10.1145/359657.359659>
50. Panti, M., Spalazzi, L., Tacconi, S.: Using the NuSMV model checker to verify the Kerberos protocol (2002)
51. Pasareanu, C.S., Giannakopoulou, D., Bobaru, M.G., Cobleigh, J.M., Barringer, H.: Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. Formal Methods Syst. Des. **32**(3), 175–205 (2008). <https://doi.org/10.1007/s10703-008-0049-6>

52. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Apt, K.R. (ed.) *Logics and Models of Concurrent Systems*. NATO ASI Series, vol. 13, pp. 123–144. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-82453-1_5
53. Schmidt, B., Meier, S., Cremers, C.J.F., Basin, D.A.: Automated analysis of Diffie-Hellman protocols and advanced security properties. In: Chong, S. (ed.) *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, 25–27 June 2012*, pp. 78–94. IEEE Computer Society (2012). <https://doi.org/10.1109/CSF.2012.25>
54. Whitefield, J., Chen, L., Sasse, R., Schneider, S., Treharne, H., Wesemeyer, S.: A symbolic analysis of ECC-based direct anonymous attestation. In: *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, 17–19 June 2019*, pp. 127–141. IEEE (2019). <https://doi.org/10.1109/EuroSP.2019.00019>
55. Woo, T.Y.C., Lam, S.S.: Authentication for distributed systems. *Computer* **25**(1), 39–52 (1992)
56. Woo, T.Y., Lam, S.S.: A semantic model for authentication protocols. In: *Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, pp. 178–194. IEEE (1993)
57. Zhang, Z., de Amorim, A.A., Jia, L., Păsăreanu, C.: Automating compositional analysis of authentication protocols. In: *2020 Formal Methods in Computer Aided Design, FMCAD 2020* (2020)

Modelling and Composition



Component-Based Approach Combining UML and BIP for Rigorous System Design

Salim Chehida^(✉) , Abdelhakim Baouya , and Saddek Bensalem 

University of Grenoble Alpes, CNRS, VERIMAG, 38000 Grenoble, France
{salim.chehida, abdelhakim.baouya, saddek.bensalem}@univ-grenoble-alpes.fr

Abstract. The development of critical systems requires the definition of a rigorous design approach enabling to check these systems before a real operation. This paper presents a component-based approach that combines UML and BIP languages for the specification and formal verification of systems. We begin by modeling the system architecture and behavior using UML. The UML models are then translated into a formal specification expressed in BIP. Finally, the SBIP framework is used for verifying the correctness of the system using Statistical Model Checking while satisfying requirements expressed by temporal properties. We apply our approach to a dam infrastructure, represented by a set of deployed sensors.

Keywords: UML · BIP · Statistical Model Checking · Sensors · Stochastic Behavior · Authentication

1 Introduction

Managing complexity, increasing trust, and promoting automation are the major challenges facing developers of critical systems. Model-based design is one of the effective solutions to address these challenges by facilitating system modeling through multiple abstractions and enabling the integration of techniques and tools for system verification.

In this work, we propose a component-based approach based on a graphical representation in the Unified Modeling Language (UML) [26], and BIP (Behavior, Interaction, Priority) [4], a highly expressive component-based language for the formal design of systems. Integrating UML and BIP is an appropriate way for the rigorous development of complex and critical systems. On one hand, UML is a standard graphical notation that has a visual and structural aspect through its diagrams. On the other hand, BIP is a textual representation that allows building formal models with the support of external code for specifying component behaviors. Moreover, BIP has stochastic semantics and an efficient tool for analysis based on statistical model checking techniques.

The approach declines in three steps. First, in Sect. 2, we use UML component diagrams to describe the architecture of the system by specifying the components

and their relationships, then UML state machine diagrams to express the behavior of each component. Secondly in Sect. 3, we translate UML diagrams into a formal representation in BIP. Finally in Sect. 4, we apply the *SBIP* tool [21] for simulating the BIP model and analyzing the system behavior using Statistical Model Checking (SMC). We check that the system satisfies some formal requirements using a set of quantitative and qualitative properties expressed in LTL (Linear-time Temporal Logic) [24] based on model simulations. We rely on the industrial case study of sensors system from the Cecebre dam infrastructure in the city of “la Coruña” in Spain. To sum up, Sect. 5 identifies the related work and we draw our conclusions and perspectives in Sect. 6.

2 System Modeling with UML

To model our system architecture and behavior, we use respectively component diagram and state machine diagrams. We use the Eclipse Papyrus Tool¹ adopted in our project to build the UML diagrams. We first present our case study.

2.1 Case Study

As shown in Fig. 1, the dam infrastructure we consider in this work is equipped with wireless sensors that collect and report sensed data to a platform called SICA-MEDUSA.

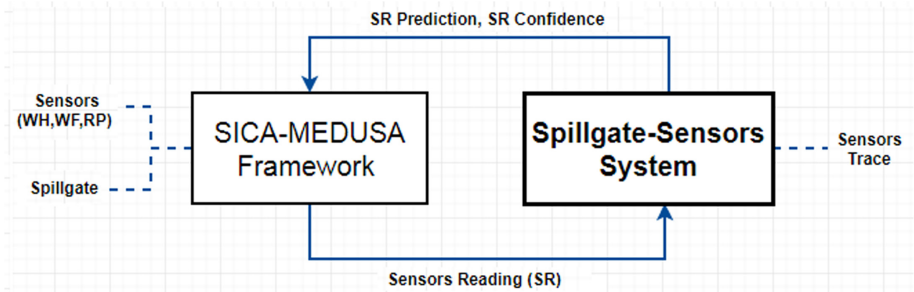


Fig. 1. Overview of the dam system

The sensors measure the Water Height (WH), the Rain Precipitation (RP), and the Water Flow (WF) in the dam. The data collected by SICA-MEDUSA help to control the opening of the spillgate and avoid the dam overflow. In this work, we aim to build a system for evaluating the confidence of sensors readings (i.e., if sensors readings are rare, possible, etc.) for a given day and making predictions for the next day after the authentication of sensors data received

¹ <http://www.eclipse.org/papyrus/>.

from SICA-MEDUSA. Our system helps for the management of the spillgate. To build our system, we rely on a trace of data recorded by each sensor per day for 28 years (from 1989 to 2016). In the following sections, we will explain how to build behavioral models for predicting sensors readings and calculating their confidence from this trace.

2.2 Architecture Model

UML component diagram is used to specify the architecture of systems by defining component choices and their dependencies. Each component is represented by a rectangle with the component name, the stereotype text and icon. A port of the component is symbolized by a small square that specifies an interaction point between the component and the environment. A connection between components defines that a component provides the services that other components require. As shown in Fig. 2, our system is represented by one component (*SpillgateSensorsSystem*) structured into three subcomponents (*Auth*, *SensorReadingPredictor*, and *SensorReadingConfidence*).

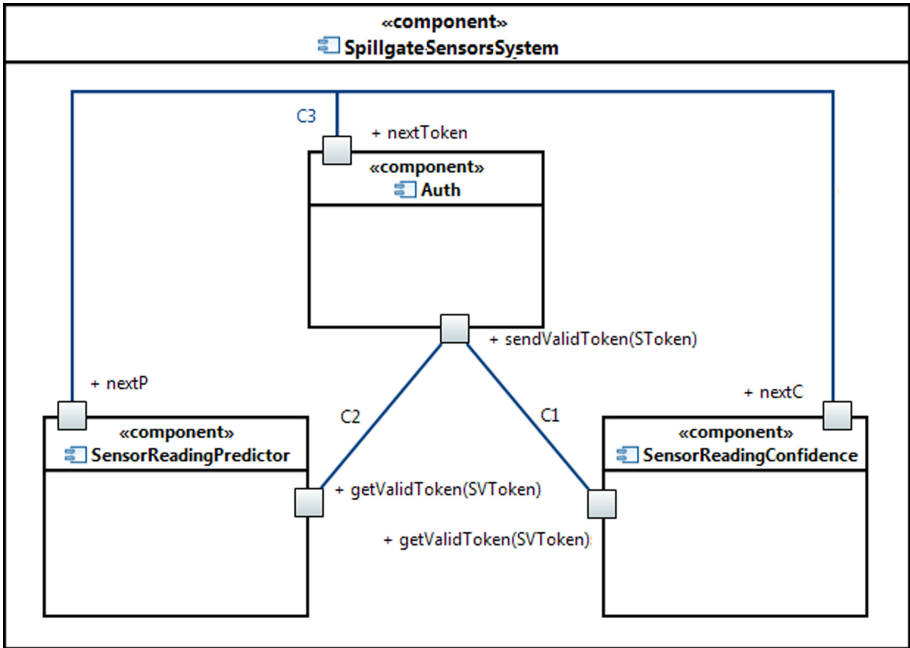


Fig. 2. Architecture of sensors system

The *Auth* component parses and authenticates the sensors data received from SICA-MEDUSA platform using JWT (JSON Web Token). JWT is an authentication system for data transferred between two parties. It represents data as a

JSON object that is encoded using a technique defined in the standard RFC7519 [15] of IETF (Internet Engineering Task Force). The JSON token consists of three parts. As shown in the example of Fig. 3, the first part, called *Header*, defines the type of the token and the hash algorithm used for its signature. The second part, called *Payload*, stores the data to be transmitted such as the token issuer (*iss*), the token expiration time (*exp*), the intended recipient of the token (*aud*), the party that the token carries information about (*sub*), and sensors information (*sensorId*, *sensorType*, *sensorReading*, and *sensingTime*). The last part of the token is the *Signature* that ensures the integrity of the data. In our system, we use the hash-based message authentication code (HMAC) algorithm to generate the signature from the encoded header and payload in base64 with a secret key using the cryptographic hash function SHA-256. The secret key prevents hackers to modify the data and produce a valid signature. As shown in Fig. 2, the *Auth* component sends the tokens to *SensorReadingPredictor* and *SensorReadingConfidence* components after validation through the connectors *C2* and *C1*.

```
{
  "typ": "JWT",
  "alg": "HS256"
}
{
  "iss": "Online JWT Builder",
  "exp": 1612268278,
  "aud": "Spillgate-Sensors-System",
  "sub": "SICA-MEDUSA",
  "sensorId": "12",
  "sensorType": "WH",
  "sensorReading": "34",
  "sensingTime": "15",
}
{
  Signature
}
```

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJpbmtpbmUgSldUIEJ1aWxkZXIiLCJpYXQiOiJlMjE2MTA3MTMwNzgsImV4cCI6IjE2MjYxMjI0ODI3OCwiYXVkaWoiOiU3BpbGxhbnYXR1bWVudCnMtU3lzdGVtIiwic3ViIjoiU0lDQ51NRURVU0EiLCJzZW5zb3JJZCI6IjEyIiwic29yVHlwZSI6IiIiwic2Vuc29yUmVhZGluZyI6IjM0Iiwic2Vuc2luZ1RpbWUiOiI1NSJ9.OYe3I21MrLMt2xgnY5RT9Iftmh0QWCaaN_7Y31I6s2I
```

Fig. 3. Example of JWT

The *SensorReadingPredictor* component gets the sensing time (*Day*) from the token and predicts the sensor reading level (*PSRL*) for the next day based on the random variable associated with a probability distribution defined from the trace of sensors data. The *SensorReadingConfidence* component gets the sensing day and sensor reading (*SR*) from the token and evaluates the confidence of this value (*SRC*) based on sensor trace. The connector *C3* allows *Auth* component proceeds to the next token after calculating *SRC* and *PSRL*.

2.3 Behavior Models

We use the UML state machine diagram to specify the behavior of the system components. State machine diagrams allow describing the states of an object or a component while interacting with other objects, components, or actors. They are represented by a state graphs connected by directed arcs that describe the transitions. A state is a situation in the life of a component during which the component satisfies a condition, performs an activity, or waits for an event. A transition allows a component to change state. It has an event, a condition (called guard) and an action denoted as “*event [condition]/action*”. An event is an instant in time that can be significant for the behavior of the component. A guard is evaluated when the event occurs. An action consists in the invocation of an operation if the event occurs and if the condition is true.

Figure 4 shows the behavior of *Auth* component of Fig. 2. An initial pseudostate is shown as a small solid filled circle. After receiving the JWT (SToken) from SICA-MEDUSA, the function *checkToken* tests if this token is well-formed and was signed by the sender and not altered in any way. If so, it will be sent to the other components, otherwise, we switch to the next token. Papyrus tool allows to define the functions *checkToken* and *getToken* using a programming language like C++.

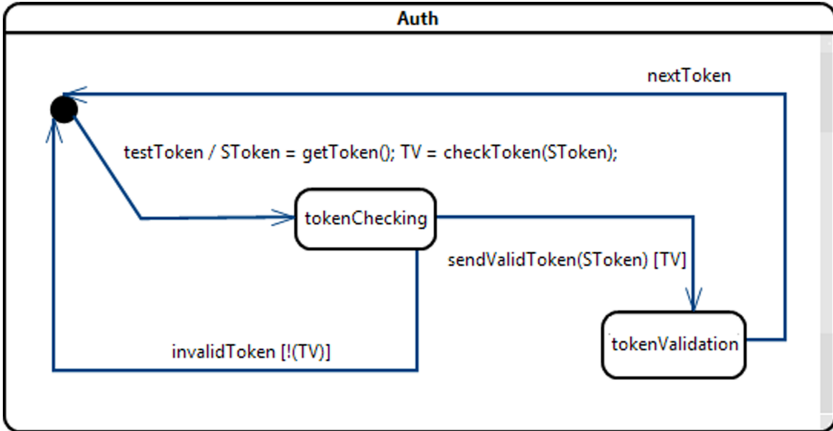


Fig. 4. The behavior of *Auth* component

Figure 5 presents the stochastic behavior of *SensorReadingPredictor* component. In [7,8], we explain how to specify this behavior starting from a given trace of sensors data. After cleaning, we discretize sensor data into five fixed levels using EWD (Equal Width Discretization) method [12]. Then, we generate a sensor distribution file for each day by counting the occurrence of each level of sensor readings this day. As shown in Fig. 5, when receiving the valid token with the sensor data for any day, the function *getSensorDist* selects the

corresponding distribution file (*DSD*) of a given sensor for the next day. The predictive sensor reading level (*PSRL*) is defined based on *DSD* distribution. We use a special way to represent this stochastic behavior in BIP by associating the distribution with a random variable (see Listing 1.3).

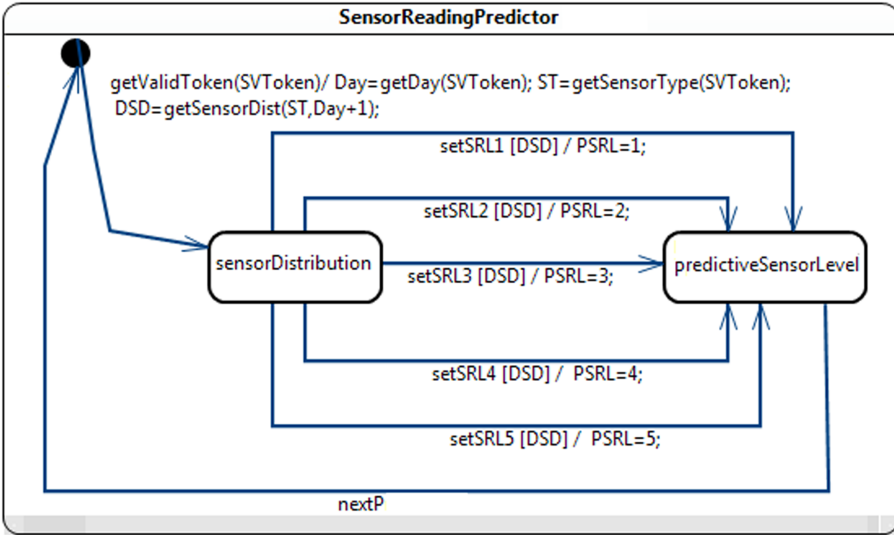


Fig. 5. The behavior of *SensorReadingPredictor* component

In Fig. 6, we specify the behavior of *SensorReadingConfidence* component. The function *discrete* calculates the sensor reading level (*SRL*) from the token. Then, the sensor reading confidence (*SRC*) is decided according to the results of functions *isVeryPossible* (observed more than 21 times in 28 years for a given day), *isPossible* (observed 3 to 21 times in 28 years for a given day), *isRare* (observed once or twice within 28 years for a given day), and *isNotObserved* (never seen in 28 years for a given day) that allow respectively to check if *SRL* is very possible, possible, rarely observed or never observed for a given day from the trace of sensors data.

3 From UML to BIP

The objective of the translation of UML component and state machine diagrams into BIP specification is to check our system by taking advantage of rigorous reasoning and verification tool (*SBIP*) that supports BIP. The choice of the translation to BIP is also motivated by its capability to use external code for specifying stochastic behaviors and security features needed for our dam system.

BIP (Behavior, Interaction, and Priority) [4] is a highly expressive component-based language for the rigorous design of complex and critical systems. It allows

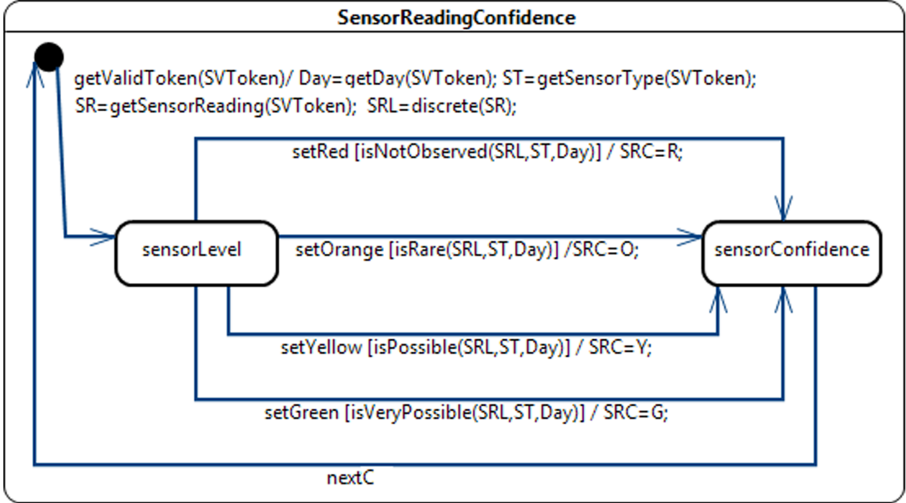


Fig. 6. The behavior of *SensorReadingConfidence* component

representing the behavior of systems using a set of components, a set of interactions that defines the possible interactions between the components, and a set of priorities for defining interaction schedule policies. BIP supports the specification of composite, hierarchically structured components, called *Compounds*, starting from the atomic ones. In BIP, the atomic components, called *Atoms*, are finite-state automata having transitions labeled with ports and states (or places) that denote control locations where the component waits for interactions. Ports are actions that can be associated with data stored in local variables and used for interactions with other components. Connectors relate ports from components by assigning them to a synchronization attribute, which may be either trigger or synchronous. A compound type defines a level of the hierarchy. It contains instances of component and connector types with connection definitions and also priorities to schedule the interactions between these components.

In this section, we will translate the UML component and state machine diagrams given in Sect. 2 into BIP. Table 1 defines the main mapping rules to translate the basic structures needed for our case study from UML to BIP. Work in progress considers other structures such as the priorities of interactions between components. A prototype is developed to automate the translation using the Eclipse Acceleo Tool². In the future, we plan to prove the correctness of the translation made by our tool.

The UML composite components become BIP compounds and the UML atomic components become BIP atoms. Connections between UML components are represented by BIP connectors. State machine diagrams specifying the behavior of UML atomic components are also translated into BIP specification attached

² <https://www.eclipse.org/acceleo/>.

Table 1. Transformation of basic structures

UML	BIP
Composite component	Compound
Atomic component	Atom
Connection	Connector
State	Place
Transition Event	External Port/Internal Port
Transition action	BIP expression
Transition guard	BIP guard expression (provided)
Variable	BIP Data

to the corresponding BIP atoms. States are specified by BIP places and transition events by BIP ports. Then, we associate the transition actions and transition guards to the BIP ports. More details and descriptions will be given in the following paragraphs.

Listing 1.1 shows the translation of the UML diagrams presented in the previous Section into BIP code. Among the advantages of BIP are the capability to use external functions and data types defined using a programming language like C++ (See the BIP documentation³ for more details). In this work, we define the external functions to be called by our BIP code in the C++ file *senSysFunc.cpp*. These are the functions called by the transitions actions or transitions guards in UML behavior models of Figs. 4, 5 and 6. In line 1 of Listing 1.1, we import the external C++ file. Functions can be defined in UML state machine diagrams using the Papyrus tool and then integrated into the external C++ file when generating the BIP specification.

In line 2, our system is represented by the BIP package *SensorsSystem* contained in a single file. Lines 4 to 16 declare the external functions invoked by the components *Auth* (see Fig. 4), *SensorReadingPredictor* (see Fig. 5), and *SensorReadingConfidence* (see Fig. 6). Line 17 declares the external data *distribution.t* needed to specify the stochastic behavior of *SensorReadingPredictor* component.

As mentioned earlier, the transition events in UML state machine diagrams are represented by ports in BIP. With BIP, we can define types of ports. Our system involves two types of ports. In line 19, the type *port0* represents events without arguments such as event *testToken* in the behavior model of *Auth* component (Fig. 4). The type *port1* in line 20 for events that have one *string* argument such as event *sendValidToken* in Fig. 4.

As for the ports, BIP allows defining types of connectors to characterize the interactions between components. A connector type in BIP is characterized by: a type name, a list of port parameters, an exported port (if any) and a set of actions for data transfers.

³ <https://www-verimag.imag.fr/TOOLS/DCS/bip/doc/latest/html/index.html>.

Listing 1.1. BIP code of the main system component

```

1  @cpp(src="ext-cpp/senSysFunc.cpp",include="senSysFunc.cpp")
2  package SensorsSystem
3  ///--- external functions and data
4  extern function string getToken()
5  extern function bool checkToken(string)
6  extern function int getDay(string)
7  extern function string getSensorType(string)
8  extern function float getSensorReading(string)
9  extern function int discrete(float)
10 extern function bool isNotObserved(int,string,int)
11 extern function bool isRare(int,string,int)
12 extern function bool isPossible(int,string,int)
13 extern function bool isVeryPossible(int,string,int)
14 extern function string getSensorDist(string,int)
15 extern function int select(distribution_t,int)
16 extern function distribution_t init_distribution(string,int)
17 extern data type distribution_t
18 // --- Ports and connectors types definitions
19 port type port0()
20 port type port1(string px)
21 connector type connect0(port1 p1, port1 p2)
22 define p1 p2 on p1 p2 down { p2.px= p1.px; } end
23 connector type connect1(port0 p1, port0 p2, port0 p3)
24 define p1 p2 p3 end
25 // --- Atom types definitions
26 atom type Auth()
27 //.. see Listing 1.2
28 end
29 atom type SensorReadingPredictor()
30 //.. see Listing 1.3
31 end
32 atom type SensorReadingConfidence()
33 //..
34 end
35 ///--- Compound types definitions
36 compound type SpillgateSensorsSystem()
37 component Auth Auth1()
38 component SensorReadingPredictor SRP1()
39 component SensorReadingConfidence SRC1()
40 ///--- Connector instantiations
41 connector connect0 C2(Auth1.sendValidToken, SRP1.getValidToken)
42 connector connect0 C1(Auth1.sendValidToken, SRC1.getValidToken)
43 connector connect1 C3(Auth1.nextToken, SRP1.nextP, SRC1.nextC)
44 end
45 end

```

In lines 21 and 22, we define the connector type *connect0* that takes as parameters two ports *p1* and *p2* of types *port1*. The *connect0* type is used for describing the actions performed when the connections *C1* and *C2* in UML architecture model of Fig. 2 happen. These connections allow sending the validated token from *Auth* component to *SensorReadingPredictor* and *SensorReadingConfidence* components. In lines 23 and 24, the *connect1* type allows to describe synchronization link between three ports such as the ports *nextP*, *nextC* and *nextToken* of type *port0* in Fig. 2 (connector *C3*). Lines 26 to 34 define the three atomic components of our system (see Listing 1.2 and 1.3 for more detail). In lines 36 to 44, we create a BIP compound that composes the three atoms. In the compound, we instantiate the components then connectors by giving the exported ports of components.

The BIP atom *Auth* in Listing 1.2 is defined from the state machine diagram of Fig. 4. The *SToken* and *TV* variables handled by the transitions actions and transitions guards in the UML diagram are declared as data in lines 2 and 3. The transition guards are declared as BIP ports. The ports *sendValidToken* and *nextToken* preceded by “*export*” (lines 5 and 6) allow *Auth* atom to communicate with the other components. The ports *testToken* and *invalidToken* in line 4 (called silent) allow to run internal actions. In line 7, UML states are represented by BIP places. Lines 8 to 14 describe the transition between states (*from .. to ..*), the transitions guards (*provided*) and the transitions actions (*do { }*) associated with each BIP port.

Listing 1.2. BIP code of *Auth* atom

```

1  atom type Auth()
2  data string SToken
3  data bool TV
4  port port0 testToken, invalidToken
5  export port port1 sendValidToken(SToken)
6  export port port0 nextToken
7  place START, tokenChecking, tokenValidation
8  initial to START
9  on testToken from START to tokenChecking
10 do {SToken = getToken(); TV = checkToken(SToken);}
11 on sendValidToken from tokenChecking to tokenValidation
12 provided (TV)
13 on invalidToken from tokenChecking to START provided (!TV)
14 on nextToken from tokenValidation to START
15 end

```

In the same way, the state machine diagram of Fig. 5 describing the behavior of *SensorReadingPredictor* atom is translated into BIP code of Listing 1.3. The connector *C2* defined in Listing 1.1 allows to joint the port *getValidToken* and recover valid token (*SVToken*) from *Auth* component.

As mentioned in Sect. 2.3, we use a special way to translate the stochastic behavior from UML state machine diagram to BIP. In line 7, we use the external

data type *distribution_t* declared in line 17 of Listing 1.1 to express the stochastic behavior of the atom. In line 18, the external functions *init_distribution* and *getSensorDist* declared in Listing 1.1 get and initiate the corresponding distribution (*DSD*). We add an integer random variable *x* in line 6 then we associate this variable with the *DSD* distribution in line 19 using the predefined function *select*. In lines 21 to 30, the internal ports *setSRL1*, *setSRL2*, *setSRL3*, *setSRL4*, *setSRL5* calculate the predictive sensor reading level (*PSRL*) based on value of the random variable *x* associated with the *DSD* distribution. In line 31, the external port *nextP* switches the execution to *Auth* atom through *C3* connector.

For more details about the specification of stochastic behaviors with BIP and their analysis, refer to [20].

Listing 1.3. BIP code of SensorReadingPredictor atom

```

1  atom type SensorReadingPredictor()
2  data string SVToken
3  data int Day
4  data string ST
5  data int PSRL
6  data int x
7  data distribution_t DSD
8  data int size
9  port port0 setSRL1, setSRL2, setSRL3, setSRL4, setSRL5
10 export port port1 getValidToken(SVToken)
11 export port port0 nextP
12 place START, sensorDistribution, predictiveSensorLevel
13 initial to START
14 on getValidToken from START to sensorDistribution
15 do {
16   Day = getDay(SVToken);
17   ST = getSensorType(SVToken);
18   DSD = init_distribution(getSensorDist(ST,Day+1),size);
19   x = select(DSD,size);
20 }
21 on setSRL1 from sensorDistribution to predictiveSensorLevel
22 provided (x==0) do { PSRL =1;}
23 on setSRL2 from sensorDistribution to predictiveSensorLevel
24 provided (x==1) do { PSRL =2;}
25 on setSRL3 from sensorDistribution to predictiveSensorLevel
26 provided (x==2) do { PSRL =3;}
27 on setSRL4 from sensorDistribution to predictiveSensorLevel
28 provided (x==3) do { PSRL =4;}
29 on setSRL5 from sensorDistribution to predictiveSensorLevel
30 provided (x==4) do { PSRL =5;}
31 on nextP from predictiveSensorLevel to START
32 end

```

The BIP atom code corresponding to state machine diagram of Fig. 6 is defined in the same way as *SensorReadingPredictor* and *Auth* atoms.

4 System Simulation and Verification

The BIP model built from UML diagrams gives a precise semantics of our system and makes its simulation and formal verification possible. In this work, we use a verification technique called SMC (Statistical Model Checking) to check our system. This technique is scalable and less memory intensive compared to model checking [2]. SMC uses a simulation-based approach to reason about formal requirements expressed in temporal logic properties. Using SMC, executions are first sampled, after which statistical techniques are applied to answer two types of questions:

1. *Quantitative*: what is the probability that the system S satisfies a given property ϕ ?
2. *Qualitative*: is the probability of a given property ϕ being satisfied by the system S is greater or equal to a certain threshold θ ?

Several SMC tools have been proposed and applied for the analysis of various case studies. In this study, we use the SBIP Statistical Model Checking tool [21] that supports the BIP language for the verification and analysis of our system. SBIP⁴ has a graphical user-interface permitting to edit, compile and simulate BIP models, and automates the different statistical analysis. It also provides a summary of the performed analysis and generates specific curves and/or plots of results. SBIP allows to express properties using a stochastic bounded variant of LTL (Linear-time Temporal Logic) [24]. It is an extension of LTL where temporal operators can be bounded and formulas can be preceded by a probabilistic operator P . In LTL, path formulas are defined using four bounded temporal operators namely, **Next** ($N\psi_1$), **Until** ($\psi_1 \cup^k \psi_2$), **Eventually** ($F^k\psi_1$), and **Always** ($G^k\psi_1$), where k is an integer value that specifies the length of the considered system execution trace and ψ_1, ψ_2 are called state formulas, which is a Boolean predicate evaluated on the system states. More details on the different LTL operators can be found in [24]. SBIP makes it possible to express and check parametric property $\phi(\mathbf{x})$, where \mathbf{x} is a parameter ranging over a finite instantiation domain.

In this work, we defined a set of LTL properties expressed on the BIP model presented in Sect. 3. We present three examples of properties to test our system on the sensed data for 2017. The three properties were evaluated in a few seconds.

PR1: Check whether the tokens received by the system in 2017 are valid.

In LTL: $P_{>=0.99}[F^{10000}(TV = true \wedge Day = D)]; \quad D = 1 : 366 : 1;$

The results are given in Fig. 7. They show that all the tokens received by *Auth* component from SICA-MEDUSA platform in 2017 are well-formed and

⁴ <http://www-verimag.imag.fr/BIP-SMC-A-Statistical-Model-Checking.html?lang=en>.

have a valid signature. *Pr1* is used for checking the authentication and integrity of sensed data. With this property, it is possible to detect if a token has been modified or sent by a user other than SICA-MEDUSA.

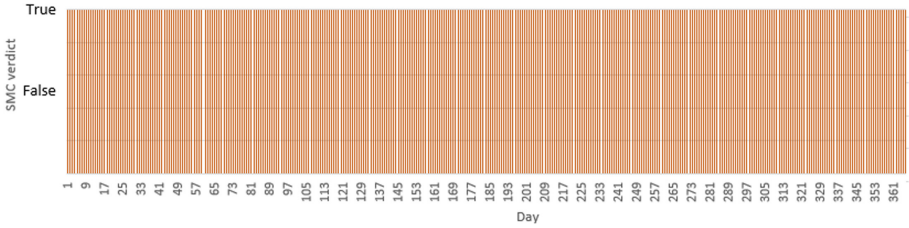


Fig. 7. Probability that received tokens in 2017 are valid

PR2: Compute the probability of predictive levels of *WH*, *WF* and *RP* sensors on January 27.

In LTL:

$$\left\{ \begin{array}{l} P_{=?}[F^{10000} (ST = T \wedge PSRL = L \wedge Day = 26)]; \quad L = 1 : 5 : 1; \\ T \in \{WH, WF, RP\}; \end{array} \right.$$

Figure 8 presents the results given by SBIP for *PR2*. We see that level 1 is the most likely for both *RP* and *WF* sensors with a probability of more than 0.80 and levels 3, 4, and 5 are never observed on this day. For *WH* sensor, level 3 is most likely, the levels 2 and 4 are less likely, and levels 1 and 5 are never observed on this day. The predictions of the sensors' levels can help to manage the dam spillgate.

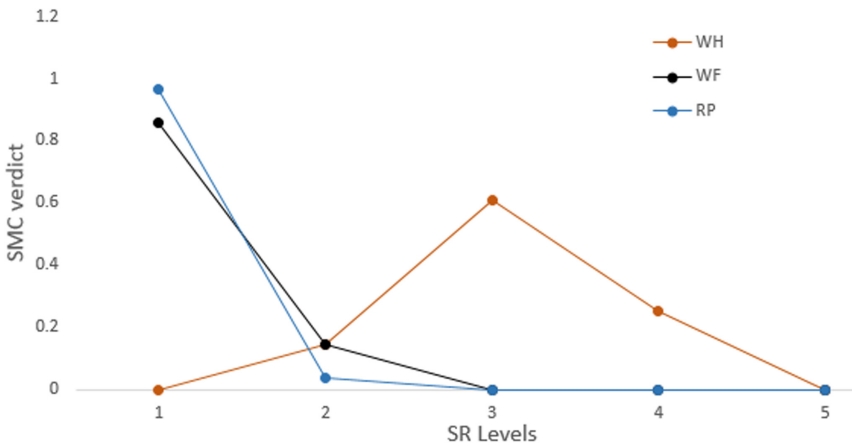


Fig. 8. Probability of *WH*, *WF* and *RP* predictive levels on January 27

PR3: Check the absence of rare levels of sensors readings in February 2017.

In LTL:

$$\left\{ \begin{array}{l} P_{>=0.99}[F^{10000} (ST = T \wedge \neg(SRC = O) \wedge Day = D)]; \quad D = 32 : 59 : 1; \\ T \in \{WH, WF, RP\}; \end{array} \right.$$

The SMC verdict of *PR3* in Fig. 9 shows two levels rare of *RP* sensor recorded on February 2nd and 3rd. For *WH* and *WF* sensors, no rare levels are detected in February 2017. In the same way, we can define properties to check for the absence of levels never observed for a given period.

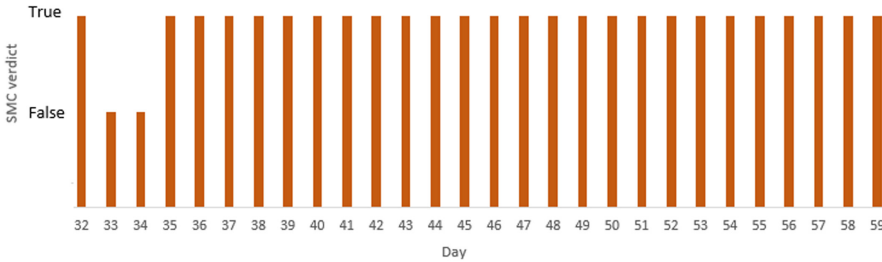


Fig. 9. Probability that RP level is not rarely observed in February 2017

5 Related Work

Several works have been proposed for coupling between UML and formal methods such as Z [13], Object-Z [16], and Event-B [25]. The works around the B method [1] are numerous. The paper [6], for instance, proposes an approach that uses UML activity diagrams for modeling system behavior considering access control policy based on RBAC (Role-Based Access Control) model, then translates UML diagrams into B in order to validate the RBAC policy using animation.

In our work, we combine UML and formal verification tools to check if the system satisfies some properties. Several works have proposed approaches to Model Checking UML state machines using the model checkers UPPAAL [17], PAT [28], USM²C [19] and PRISM [3]. In [23], the authors propose the tool TANGRAM for modeling, and verifying component-based real-time systems. The tool translates UML component and state machine diagrams into timed automata. The UPPAAL model checker is then applied to verify system correctness. The paper [11] translates UML activity diagrams extended for embedded systems into the front-end languages used by the model checkers NuSMV, SPIN, UPPAAL and PES, then compares the performance of different model checking tools.

In our approach, we apply Statistical Model Checking (SMC) after translating UML component and state machine diagrams into a formal specification

expressed in BIP. SMC technique has been proposed to improve scalability by combining simulation and statistical methods to reason about properties. It is used as an alternative to time and memory intensive techniques like model checkers. The paper [2] presents a survey of SMC tools. For instance, PRISM-SMC [18] implements SMC techniques such as Probability Estimation techniques (PE) and Hypothesis Testing, and the model to be checked is constructed before and stored in memory. MRMC [22] offers SMC with confidence interval computation. However, it always loads Markov chain representations into memory completely. UPPAAL-SMC [10] and Ymer [27] are closer to SBIP, and both of them consider GSMP (Generalized Semi Markov Processes). UPPAAL-SMC provides a general stochastic timed semantics and is limited to exponential and uniform density functions. Furthermore, BIP is a component-based language endowed with capabilities to express automata-based and/or Petri Net behavior and to call external code. To perform SMC, the BIP engine relies on the constructed models in C++.

There are also some works that propose the integration of other languages with BIP. For instance, [5] proposes a security-based modelling language for representing data access controls in IoT systems and implements a tool to translate IoT models into BIP. The BIP generated model is used to simulate the system and test if the security policy is guaranteed. The paper [14] presents an approach for the translation from the robotic framework GenoM3 to the real-time extension of BIP. After the transformation, the BIP model augmented with a timed-property monitor is executed to check properties online and react in case of violation. The paper [9] provides a general methodology and tool for translating AADL models into BIP models in order to enable the simulation of AADL models, as well as application of verification techniques, such component-based deadlock detection.

6 Conclusion

In this paper, we have proposed a component-based approach to support the rigorous design of software systems and have applied this approach to a dam system. System components with their interactions and behaviors are modeled with UML component and state machine diagrams. UML models are translated into a formal specification expressed in BIP. Finally, SBIP is used to simulate the BIP specification and check system requirements expressed by LTL. The main benefits of our approach are:

- It supports the specification of composite components, hierarchically structured from atomic components, which facilitates reusability and maintainability of the system components.
- UML graphical models facilitate the understanding of the system by designers without background in formal methods.
- BIP allows the rigorous specification of the system and offers the possibility of calling external code to specify different types of components such as stochastic and security components.

- We use SMC, a scalable and less memory-intensive technique compared to model checking, for verifying system requirements. Verification helps mitigate the risk of software errors.

We are planning in the future to work in two directions: (*i*) validating the translation of UML models into BIP using proof techniques, (*ii*) generating Java code from BIP and wrapping the code in an envelope called bundles to enable its dynamic deployment in execution platforms such as OSGi.

Acknowledgments. The research leading to these results has been supported by the European Union through the BRAIN-IoT project (Grant agreement ID: 780089) and the CPS4EU project (Grant agreement ID: 826276). The authors would like to thank EMALCSA Company for the data collected from the dam infrastructure.

References

1. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Agha, G., Palmkog, K.: A survey of statistical model checking. *ACM Trans. Model. Comput. Simul.* **28**(1), 1–39 (2018). <https://doi.org/10.1145/3158668>
3. Baouya, A., Bennouar, D., Mohamed, O.A., Ouchani, S.: A probabilistic and timed verification approach of SysML state machine diagram. In: 2015 12th International Symposium on Programming and Systems (ISPS) (2015). <https://doi.org/10.1109/ISPS.2015.7245001>
4. Basu, A., et al.: Rigorous component-based system design using the BIP framework. *IEEE Softw.* **28**(3), 41–48 (2011)
5. Beaulaton, D., et al.: A language for analyzing security of IOT systems. In: 2018 13th Annual Conference on System of Systems Engineering (SoSE), pp. 37–44 (2018)
6. Chehida, S., Idani, A., Ledru, Y., Kamel Rahmouni, M.: Combining UML and B for the specification and validation of RBAC policies in business process activities. In: 2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS), pp. 1–12 (2016). <https://doi.org/10.1109/RCIS.2016.7549284>
7. Chehida, S., Baouya, A., Bensalem, S., Bozga, M.: Applied statistical model checking for a sensor behavior analysis. In: Shepperd, M., Brito e Abreu, F., Rodrigues da Silva, A., Pérez-Castillo, R. (eds.) QUATIC 2020. CCIS, vol. 1266, pp. 399–411. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58793-2_32
8. Chehida, S., Baouya, A., Bensalem, S., Bozga, M.: Learning and analysis of sensors behavior in iot systems using statistical model checking. *Softw. Qual. J.* **2020**, 1–22 (2021). <https://doi.org/10.1007/s11219-021-09559-w>
9. Chkouri, M.Y., Robert, A., Bozga, M., Sifakis, J.: Translating AADL into BIP - Application to the verification of real-time systems. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 5–19. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01648-6_2
10. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B.: UPPAAL SMC tutorial. *Int. J. Softw. Tools Technol. Transf.* **17**(4), 397–415 (2015). <https://doi.org/10.1007/s10009-014-0361-y>
11. Daw, Z., Cleaveland, R.: Comparing model checkers for timed UML activity diagrams. *Sci. Comput. Program.* **111**, 277 (2015)

12. Dougherty, J., Kohavi, R., Sahami, M.: Supervised and unsupervised discretization of continuous features. In: Prieditis, A., Russell, S. (eds.) *Machine Learning Proceedings 1995*. Morgan Kaufmann, Burlington (1995)
13. Dupuy, S., Ledru, Y., Chabre-Peccoud, M.: An overview of RoZ?: a tool for integrating UML and Z specifications. In: Wangler, B., Bergman, L. (eds.) *CAiSE 2000*. LNCS, vol. 1789, pp. 417–430. Springer, Heidelberg (2000). <https://doi.org/10.1007/3-540-45140-4-28>
14. Foughali, M., Bensalem, S., Combaz, J., Ingrand, F.: Runtime verification of timed properties in autonomous robots. In: *2020 18th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)* (2020)
15. Jones, M., Bradley, J., Sakimura, N.: JSON Web Token (JWT). RFC 7519 (2015). <https://doi.org/10.17487/RFC7519>
16. Kim, S.-K., David, C.: Formalizing the UML class diagram using Object-Z. In: France, R., Rumpe, B. (eds.) *UML 1999*. LNCS, vol. 1723, pp. 83–98. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-46852-8_7
17. Knapp, A., Merz, S., Rauh, C.: Model checking timed UML state machines and collaborations. In: Damm, W., Olderog, E.-R. (eds.) *FTRTFT 2002*. LNCS, vol. 2469, pp. 395–414. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45739-9_23
18. 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
19. Liu, S., et al.: A formal semantics for complete UML state machines with communications. In: Johnsen, E.B., Petre, L. (eds.) *IFM 2013*. LNCS, vol. 7940, pp. 331–346. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38613-8_23
20. Mediouni, B.L.: *Modeling and Analysis of Stochastic Real-Time Systems*. Ph.D. thesis, Grenoble Alpes University, France (2019). <https://tel.archives-ouvertes.fr/tel-02305867>
21. Mediouni, B.L., Nouri, A., Bozga, M., Dellabani, M., Legay, A., Bensalem, S.: SBIP 2.0: statistical model checking stochastic real-time systems. In: Lahiri, S.K., Wang, C. (eds.) *ATVA 2018*. LNCS, vol. 11138, pp. 536–542. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_33
22. MRMC: MRMC tool (2011). <http://www.mrmc-tool.org>
23. Muniz, A.L.N., Andrade, A., Lima, G.: Integrating UML and UPPAAL for designing, specifying and verifying component-based real-time systems. *Innov. Syst. Softw. Eng.* **6**, 29–37 (2009)
24. Pnueli, A.: The temporal logic of programs. In: *18th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, USA (1977)
25. Siyuan, H., Hong, Z.: Towards transformation from UML to Event-B. In: *2015 IEEE International Conference on Software Quality, Reliability and Security - Companion*, pp. 188–189 (2015). <https://doi.org/10.1109/QRS-C.2015.39>
26. UML2: Unified Modeling Language (Version 2.5.1). Object Management Group (2017)
27. Younes, H.L.S.: Ymer: a statistical model checker. In: Etessami, K., Rajamani, S.K. (eds.) *CAV 2005*. LNCS, vol. 3576, pp. 429–433. Springer, Heidelberg (2005). https://doi.org/10.1007/11513988_43
28. Zhang, S.J., Liu, Y.: An automatic approach to model checking UML state machines. In: *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion* (2010)



Composable Partial Multiparty Session Types

Claude Stolze, Marino Miculan^(✉) , and Pietro Di Gianantonio 

Dept. of Mathematics, Computer Science and Physics, University of Udine,
Udine, Italy

{claude.stolze,marino.miculan,pietro.digianantonio}@uniud.it

Abstract. We introduce *partial sessions* and *partial (multiparty) session types*, in order to deal with open systems, i.e., systems with missing components. Partial sessions can be *composed*, and the type of the resulting system is derived from those of its components without knowing any suitable global type nor the types of missing parts. Incompatible types, due to e.g. miscommunications or deadlocks, are detected at the merging phase. We apply these types to a process calculus, for which we prove *subject reduction* and *progress*, so that well-typed systems never violate the prescribed constraints. Therefore, partial session types support the development of systems by incremental assembling of components.

Keywords: Multiparty session types · process algebras · open systems

1 Introduction

(*Multiparty*) *session types* (MPST) are a well-established theoretical and practical framework for the specification of the interactions between components of a distributed systems [10, 12, 15–17, 26]. The gist of this approach is to first describe the system’s overall behaviour by means of a *global type*, from which a local specification (*local type*) for each component can be derived. The system will behave according to the global type if each component respects its local type, which can be ensured by means of, e.g., static type checking [18, 24]. Therefore, session types support a *top-down* style of coding: first the designer specifies the behaviour from a global perspective, then the programmers are given the specifications for their modules. On the other hand, these session types do not fit well *bottom-up* programming models, where systems are built incrementally by composing existing reusable components, possibly with dynamic bindings. In these situations, components could offer “contracts” in the form of, e.g., session types; then, when these components are connected together, we would like to derive the contract for the resulting system from components’ ones. The system becomes a new component which can be used in other assemblies, and so on.

Work supported by the Italian MIUR project PRIN 2017FTXR7S *IT MATTERS (Methods and Tools for Trustworthy Smart Systems)*.

To this end, we need to infer the type for an *open* system (i.e., where some parts may be still missing) using the types of the known components, in a compositional way and without knowing any global type. This is challenging. As an example, let us consider a protocol from [10] with three participants: a server s , an authorization server a and some client c . First, s sends to c either a request to login, or to cancel. In the first case, c sends a password to a , and a sends a boolean to s (telling whether c is authorized). In the second case, c tells a to quit. Using the syntax of [10], the two server processes have the following types:

$$S_s := c \oplus \left\{ \begin{array}{l} \text{login}.a\&\text{auth}(\text{Bool}), \\ \text{cancel} \end{array} \right\} \quad S_a := c \& \left\{ \begin{array}{l} \text{passwd}(\text{Str}).s \oplus \text{auth}(\text{Bool}), \\ \text{quit} \end{array} \right\}$$

Let us suppose that we have implementations a and s for S_a, S_s . To prevent miscommunications, we would like to verify that these two processes work well together; e.g., we have to ensure that a can send the message $\text{auth}(\text{Bool})$ to s iff s is waiting for it. This corresponds to see these two processes as a single system $a|s$, and to check that $a|s$ is well-typed *without knowing the behaviour of clients*; more precisely, we have to figure out a session type for $a|s$ from S_a and S_s . This is difficult, because the link that propagates the choice made by s to a is the missing client c , so we have to “guess” its role without knowing it.

In this paper, we address this problem by introducing *partial sessions* and *partial (multiparty) session types*. Partial session types generalise global types with the possibility to type also partial (or *open*) sessions, i.e., where some participant may be missing. The key difference is that while a global type is a complete, “platonistic” description of the protocol, partial session types represent the *subjective* views from participants’ perspectives. We can *merge* two sessions with the same name but from two different “point of views”, whenever their types are *compatible*; in this case, we can compute the new, unified, session type from those of the components. In this way, we can guarantee important properties (e.g., absence of deadlocks) about partial session without knowing all participants beforehand, and without a complete global type. In fact, the distinction between local and global types vanishes: local types correspond to partial session types for sessions with a single participant, and global types correspond to *finalized* partial session types, i.e., in which no participant is missing.

Defining “compatibility” and how to merge partial session types is technically challenging. Intuitively, the semantics of a partial session type is the set of all possible execution traces (which depend on internal and external choices). We provide a merging algorithm computing a type covering all the possible synchronizations of these traces. Incompatible types, due to, e.g., miscommunications or deadlocks, are detected when no synchronization is possible. Also the notion of *progress* has to be revisited, to accommodate the case when a partial session cannot progress not due to a deadlock, but due to some missing participant.

The rest of the paper is organized as follows. In Sect. 2 we introduce a formal calculus for processes communicating over multiparty sessions. Partial session types are presented in Sect. 3, and the type system is in Sect. 4. Central to this type system is the merging algorithm, which we describe in Sect. 5. Subject

reduction and progress are given in Sect. 6. Finally, comparison with related work are in Sect. 7, and conclusions are in Sect. 8.

A prototype implementation of the merging algorithm can be found at <https://github.com/cstolze/partial-session-types-prototype>.

2 A Calculus for Processes over Multiparty Sessions

Our language for processes is inspired by [10], in turn inspired by [28]; as in those works, we consider *synchronous* communications. We note p, q, p_1, p', \dots for participant names, belonging to some set \mathfrak{P} ; \tilde{p} for a finite non-empty set of participants $\{p_1, \dots, p_n\}$. The syntax of processes is as follows:

$$P, Q, R ::= \bar{x}^{p\tilde{q}} \triangleright \text{in}_i.P \mid x^{pq} \triangleleft (P, Q) \mid \bar{x}^{p\tilde{q}}(y).P \mid x^{pq}(y).(P \parallel Q) \\ \mid \text{close}(x) \mid \text{wait}(x).P \mid (P \mid_x Q) \mid (\nu x)P \mid P + Q$$

The process $\bar{x}^{p\tilde{q}} \triangleright \text{in}_i.P$ sends label in_i in session x , as participant p , to \tilde{q} , and proceeds as P . This label is received by processes of the form $x^{qp} \triangleleft (Q_1, Q_2)$, which then proceeds as Q_i . The process $\bar{x}^{p\tilde{q}}(y).P$ creates a fresh subsession handler y , sends it to \tilde{q} , and proceeds as P . This handler is received by processes of the form $x^{qp}(y).(Q \parallel R)$ which forks a process Q (dedicated to y) in parallel to the continuation R (on x)¹. We compose the processes P and Q through session x with $P \mid_x Q$. $\text{close}(x)$ is the neutral element for \mid_x , while $\text{wait}(x).P$ closes session x when all the other participants are gone. $(\nu x)P$ is the standard restriction, and $P + Q$ is the standard non-deterministic choice.

The session name y is bound in expressions of the form $(\nu y)P$, $\bar{x}^{p\tilde{q}}(y).P$, and $x^{pq}(y).(P \parallel Q)$. Free names of a process P (noted $\text{fn}(P)$) are the set of free names of sessions appearing in P .

In order to define the operational semantics, we first introduce the usual notion of contextual equivalence.

Definition 2.1 (Contexts). $\mathcal{C}[-] ::= - \mid (\nu x)\mathcal{C}[-] \mid (\mathcal{C}[-] \mid_x P) \mid (P \mid_x \mathcal{C}[-])$

Definition 2.2 (Equivalence \equiv). *The relation \equiv is the smallest equivalence relation closed under contexts (that is, $P \equiv Q \Rightarrow \mathcal{C}[P] \equiv \mathcal{C}[Q]$) satisfying the following rules (we suppose that x, y and z are different session names):*

$$\begin{array}{ll} P \mid_x Q \equiv Q \mid_x P & (P \mid_x Q) \mid_x R \equiv P \mid_x (Q \mid_x R) \\ P \mid_x \text{close}(x) \equiv P & ((\nu x)P) \mid_z Q \equiv (\nu x)(P \mid_z Q) \quad x \notin \text{fn}(Q) \\ (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P & (P \mid_x Q) \mid_y R \equiv P \mid_x (Q \mid_y R) \quad x \notin \text{fn}(R), y \notin \text{fn}(P) \end{array}$$

We can see that processes have the structure of a commutative monoid, thus we will use $\prod_i^z P_i$ as a shorthand for $P_1 \mid_z \dots \mid_z P_n$.

¹ From a computational point of view, this “parallel input” corresponds to the programming practice to selectively share sessions between processes. This constructor allows us to enforce a discipline on the shared sessions in order to avoid deadlocks between processes. Moreover, it is motivated by connections with linear logic [10, 28].

$$\begin{array}{l}
 \bar{x}^{p\tilde{q}}(y).R \mid_x \Pi_i^x(x^{q_i P}(y).(P_i \parallel Q_i)) \xrightarrow{x:p \rightarrow \tilde{q}:\langle \cdot \rangle} (\nu y)(R \mid_y \Pi_i^y P_i) \mid_x \Pi_i^x Q_i \quad (\text{send}) \\
 \bar{x}^{p\tilde{q}} \triangleright \text{in}_j.R \mid_x \Pi_i^x x^{q_i P} \triangleleft (P_{1,i}, P_{2,i}) \xrightarrow{x:p \rightarrow \tilde{q}:\&\text{in}_j} R \mid_x \Pi_i^x P_{j,i} \quad (\text{case}) \\
 (\nu x)(\text{wait}(x).P) \xrightarrow{\tau} P \quad \text{if } x \notin \text{fn}(P) \quad (\text{wait}) \\
 P_1 + P_2 \xrightarrow{+} P_j \quad j \in \{1, 2\} \quad (\text{choice}) \\
 \\
 \frac{P \xrightarrow{x:\gamma} Q}{(\nu x)P \xrightarrow{\tau} (\nu x)Q} \quad \frac{P \xrightarrow{\alpha} Q \quad \forall \gamma, \alpha \neq x:\gamma}{(\nu x)P \xrightarrow{\alpha} (\nu x)Q} \\
 \frac{P \xrightarrow{\alpha} Q}{P \mid_x R \xrightarrow{\alpha} Q \mid_x R} \quad \frac{P \xrightarrow{\alpha} Q}{R \mid_x P \xrightarrow{\alpha} R \mid_x Q} \quad \frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha} Q}
 \end{array}$$

Fig. 1. Reduction for processes (where $\tilde{q} = \{q_1, \dots, q_n\}$ and i ranges over $1..n$).

Definition 2.3 (Reductions for processes). *The actions α for processes are defined as:*

$$\alpha ::= + \mid x : p \rightarrow \tilde{q} : \langle \cdot \rangle \mid x : p \rightarrow \tilde{q} : \&\text{in}_i \mid \tau$$

We may write $x : \gamma$ for either $x : p \rightarrow \tilde{q} : \langle \cdot \rangle$ or $x : p \rightarrow \tilde{q} : \&\text{in}_i$.

We note $P \xrightarrow{\alpha} Q$ for a transition from P to Q under the action α . This relation is defined by the rules in Fig. 1.

Note that the typing rules will ensure x does not escape its scope when reducing $(\nu x)(\text{wait}(x).P)$ into P .

Example 2.1. As a running example, let us consider three participants p, q, r . p chooses whether to send a message to r or not; this choice is communicated to r through an intermediate participant q .

$$\begin{aligned}
 P_p &:= (\bar{x}^{pq} \triangleright \text{in}_1.\bar{x}^{pr}(y).\text{wait}(y).\text{close}(x)) + (\bar{x}^{pq} \triangleright \text{in}_2.\text{close}(x)) \\
 P_q &:= x^{qp} \triangleleft (\bar{x}^{qr} \triangleright \text{in}_1.\text{close}(x), \bar{x}^{qr} \triangleright \text{in}_2.\text{close}(x)) \\
 P_r &:= x^{rq} \triangleleft (x^{rp}(y)(\text{close}(y) \parallel \text{close}(x)), \text{close}(x))
 \end{aligned}$$

Here is an example of execution:

$$\begin{array}{l}
 P_p \mid_x P_q \mid_x P_r \xrightarrow{+} \bar{x}^{pq} \triangleright \text{in}_2.\text{close}(x) \mid_x P_q \mid_x P_r \\
 \xrightarrow{x:p \rightarrow q:\&\text{in}_2} \bar{x}^{qr} \triangleright \text{in}_2.\text{close}(x) \mid_x P_r \\
 \xrightarrow{x:q \rightarrow r:\&\text{in}_2} \text{close}(x)
 \end{array}$$

Notice that p can start the session with just q and then wait for input from r :

$$\begin{array}{l}
 P_p \mid_x P_q \xrightarrow{+} \bar{x}^{pq} \triangleright \text{in}_2.\text{close}(x) \mid_x P_q \\
 \xrightarrow{x:p \rightarrow q:\&\text{in}_2} \bar{x}^{qr} \triangleright \text{in}_2.\text{close}(x)
 \end{array}$$

3 Partial Multiparty Session Types

Partial multiparty session types (or just “session types”) define the behaviour of a partial session. Their syntax is as follows (where $i \in \{1, 2\}$):

$$G ::= p \rightarrow \tilde{q} : \&\text{in}_i; G \mid p \rightarrow \tilde{q} : \langle G \rangle; G \mid G \oplus G \mid G \& G \mid \text{end} \mid \text{close} \mid 0 \mid \omega$$

The set of participant names appearing in G is denoted by $\text{fn}(G)$.

Informally, $p \rightarrow \tilde{q} : m; G$ means that the participant p sends the message m to the participants in \tilde{q} , then the session continues with G . The message $\&\text{in}_i$ is a label, while $\langle G \rangle$ is a fresh session handler of type G . end means that the session ends and the process survives, while close means that the session and the process end. $G_1 \oplus G_2$ (resp. $G_1 \& G_2$) denotes an *internal* (resp. *external*) choice. Internal choices are made by local participants of the session, contrary to external choices; notice that, in contrast with standard practice, sending or receiving a label $\&\text{in}_i$ is unrelated from the choices done with \oplus or $\&$. Finally, we add the *empty* type 0 , which denotes no possible executions (and it is the unit of \oplus), and the *inconsistent* type ω , which denotes an error in the session.

Example 3.1. Continuing our running Example 2.1, the following should be the types of each participant.

$$\begin{aligned} G_p &:= (p \rightarrow q : \&\text{in}_1; p \rightarrow r : \langle \text{end} \rangle; \text{close}) \oplus (p \rightarrow q : \&\text{in}_2; \text{close}) \\ G_q &:= (p \rightarrow q : \&\text{in}_1; q \rightarrow r : \&\text{in}_1; \text{close}) \& (p \rightarrow q : \&\text{in}_2; q \rightarrow r : \&\text{in}_2; \text{close}) \\ G_r &:= (q \rightarrow r : \&\text{in}_1; p \rightarrow r : \langle \text{close} \rangle; \text{close}) \& (q \rightarrow r : \&\text{in}_2; \text{close}) \end{aligned}$$

These types are actually assigned to P_p, P_q, P_r by the type system we will present in Sect. 4. But moreover, we would like to be able to type also compositions of these processes; e.g. the types of $P_p \mid_x P_q$ and $P_q \mid_x P_r$ should be the following:

$$\begin{aligned} G_{p,q} &:= (p \rightarrow q : \&\text{in}_1; q \rightarrow r : \&\text{in}_1; p \rightarrow r : \langle \text{end} \rangle; \text{close}) \\ &\quad \oplus (p \rightarrow q : \&\text{in}_2; q \rightarrow r : \&\text{in}_2; \text{close}) \\ G_{q,r} &:= (p \rightarrow q : \&\text{in}_1; q \rightarrow r : \&\text{in}_1; p \rightarrow r : \langle \text{close} \rangle; \text{close}) \\ &\quad \& (p \rightarrow q : \&\text{in}_2; q \rightarrow r : \&\text{in}_2; \text{close}) \end{aligned}$$

Moreover, notice that $G_{p,q}$ describes also the behaviour of $P_p \mid_x P_q \mid_x P_r$. As we will see in the next section, these types can be inferred from G_p, G_q, G_r in a compositional way. \square

A *chain of communications* is a type of the form $C_1; C_2; \dots; C_n$. Messages m and *communications* C are defined as follows:

$$m ::= \&\text{in}_i \mid \langle G \rangle \quad C ::= p \rightarrow \tilde{q} : m \mid \text{end} \mid \text{close} \mid 0 \mid \omega \mid 1$$

The communications end , close , 0 , ω are also types and they are called *terminal*; the only non-terminal communications are $p \rightarrow \tilde{q} : m$ and 1 , the latter representing any communication which is not observable from the current process.

As such, we can prefix 1 to any session type G by defining $1;G := G$. We denote by \mathfrak{C}_ω the set of all communications, and by $\mathfrak{C} = \mathfrak{C}_\omega \setminus \{\omega, 0\}$ the set of *executable* communications. We pose \mathfrak{S}_ω the set of all session types, and \mathfrak{S} the set of session types where there is no occurrence of 0 and ω . By default, we will use \mathfrak{S}_ω , while \mathfrak{S} will be used to type processes in Sect. 4.

In the following, we denote by S, S_1, \dots sets of participants, and we use the shorthand $S_1 \# S_2$ for $S_1 \cap S_2 = \emptyset$. A set of participant S will be called *viewpoint*.

Definition 3.1 (Independence relation). *We define the independence of communications relative to a set of participants S as the smallest symmetric relation I_S such that $C I_S 1$ for any C , and $(p \rightarrow \bar{q} : m) I_S (p' \rightarrow \bar{q}' : m')$ whenever $(\{p\} \cup \bar{q}) \cap (\{p'\} \cup \bar{q}') \# S$.*

Informally, $C_1 I_S C_2$ means that the common participants of C_1 and C_2 are not in S . This independence is relative to the viewpoint of S , because when $C_1 I_S C_2$, the viewpoint of S cannot discriminate between $C_1; C_2; G$ and $C_2; C_1; G$, as is shown in Eq. (1) below. In fact, we can define an equivalence relation between session types relative to S :

Definition 3.2 (Equivalence relation). *For any set of participants S , we define the relation \simeq_S on session types as the smallest congruence verifying the following properties:*

$$\begin{array}{l}
 C_1; C_2; G \simeq_S C_2; C_1; G \quad (\text{if } C_1 I_S C_2) \\
 G_1 \& (G_2 \oplus G_3) \simeq_S (G_1 \& G_2) \oplus (G_1 \& G_3) \quad G \& \omega \simeq_S G \quad G \& 0 \simeq_S 0 \\
 G_1 \& (G_2 \& G_3) \simeq_S (G_1 \& G_2) \& G_3 \quad G \& G \simeq_S G \quad G_1 \& G_2 \simeq_S G_2 \& G_1 \\
 G_1 \oplus (G_2 \oplus G_3) \simeq_S (G_1 \oplus G_2) \oplus G_3 \quad G \oplus G \simeq_S G \quad G \oplus 0 \simeq_S G \\
 C; (G_1 \& G_2) \simeq_S (C; G_1) \& (C; G_2) \quad C; \omega \simeq_S \omega \quad C; 0 \simeq_S 0 \\
 C; (G_1 \oplus G_2) \simeq_S (C; G_1) \oplus (C; G_2) \quad G_1 \oplus G_2 \simeq_S G_2 \oplus G_1
 \end{array} \tag{1}$$

We can see that the operations \oplus and $\&$, together with the constants 0 and ω , form a unital commutative semiring. We note $\bigoplus\{G_1, \dots, G_n\}$ for $G_1 \oplus \dots \oplus G_n$, and $\&\{G_1, \dots, G_n\}$ for $G_1 \& \dots \& G_n$. In particular, $\bigoplus \emptyset = 0$, and $\& \emptyset = \omega$. Equation (1) allows for the “out of order” execution of independent communications. Notice that in general $G \oplus \omega \not\simeq_S \omega$ because the behaviour of a process of type $G \oplus \omega$ is not necessarily always inconsistent.

The fact that choices $G_1 \oplus G_2$ or $G_1 \& G_2$ are unrelated from the action of sending a choice allows us to move these operators around without changing the meaning. Hence, we can consider *disjunctive normal forms* of session types.

Definition 3.3 (Disjunctive Normal Form). *A session type G is in Disjunctive Normal Form (DNF), if it is of the form $\bigoplus\{\& A_1, \dots, \& A_n\}$ with the A_i being sets of chains of communications where every message $\langle G' \rangle$ is in DNF.*

In DNF a type can be seen as a set of sets of traces (sequences of communications), the intuition being that a trace describes a single possible interaction of a process. A set of traces defines a deterministic strategy followed by a single

process P , describing how P reacts for any possible choice from other processes. A set of sets of traces describes all the possible strategies that P can follow once it has selected all its possible internal choices. So, describing a behaviour in DNF is like saying that a process P starts by anticipating all possible internal choices for all possible interactions during execution. After that, P becomes deterministic and reacts in a single possible way to communications of other processes.

The equivalence relation on types allows us to rewrite any type in a DNF.

Proposition 3.1. *For any type G and set of participants S , we can compute a G' in DNF such that $G' \simeq_S G$.*

4 Type System

In this section we introduce the type system for processes. A key point is that the type of a session are relative to the participants of that session.

Definition 4.1 (Environment). *A typing declaration for session x is a triple $x : \langle G \mid S \rangle$ where $G \in \mathfrak{G}$ and $S \subseteq \mathfrak{P}$. S is the set of local participants of x .*

An environment Γ is a finite set of typing declarations $\Gamma = x_1 : \langle G_1 \mid S_1 \rangle, \dots, x_n : \langle G_n \mid S_n \rangle$, such that x_1, \dots, x_n are all distinct.

The main differences between our environments and those in [10] are that session types replace local types, and each session is endowed with a set of local participants, in addition to its session type.

Definition 4.2 (Equivalent environments). *We define \simeq on environments as the smallest equivalence relation satisfying the following rule:*

$$\frac{\Gamma_1 \simeq \Gamma_2 \quad G_1 \simeq_S G_2}{\Gamma_1, x : \langle G_1 \mid S \rangle \simeq \Gamma_2, x : \langle G_2 \mid S \rangle}$$

The typing judgment is $P \vdash \Gamma$, whose rules are shown in Fig. 2.

Rules (*send*), (*recv*), (*sel_i*), and (*case*) deal with communication. Differently from most type systems (see e.g. [10]), the send and receive actions are typed by the same global type, and not by dual types: in our approach the duality is given by the set of participants, which is either the sender or the receiver.

Rules (*close*) and (*wait*) correspond respectively to the 1 and \perp rules in linear logic, and they both assume there is no named participant, therefore the set of inner participants in the conclusion is empty.

Rule (\oplus) types an internal choice between two processes, but this internal choice is not done for a single session but for the whole process, hence we need to add \oplus to every type. If the internal choice is irrelevant for some session x , that is, we have $x : \langle G \mid S \rangle$ in the two premises, then in the conclusion we would have $x : \langle G \oplus G \mid S \rangle$, which is equivalent to the former. We can of course rewrite types into equivalent ones with rule (\simeq).

Rule (ν) allows us to create a local, restricted session. To correctly type the local session, we need to check that the its type is complete, since no other

$$\begin{array}{c}
\frac{P \vdash \Gamma, y : \langle G_1 \mid \{p\} \rangle, x : \langle G_2 \mid \{p\} \rangle}{\bar{x}^{p\tilde{q}}(y).P \vdash \Gamma, x : \langle p \rightarrow \tilde{q} : \langle G_1 \rangle; G_2 \mid \{p\} \rangle} \text{ (send)} \\
\\
\frac{P \vdash \Gamma_1, y : \langle G_1 \mid \{q\} \rangle \quad Q \vdash \Gamma_2, x : \langle G_2 \mid \{q\} \rangle \quad q \in \tilde{q}}{x^{qp}(y).(P \parallel Q) \vdash \Gamma_1, \Gamma_2, x : \langle p \rightarrow \tilde{q} : \langle G_1 \rangle; G_2 \mid \{q\} \rangle} \text{ (recv)} \\
\\
\frac{P \vdash \Gamma, x : \langle G \mid \{p\} \rangle}{\bar{x}^{p\tilde{q}} \triangleright \text{in}_i.P \vdash \Gamma, x : \langle p \rightarrow \tilde{q} : \&\text{in}_i; G \mid \{p\} \rangle} \text{ (sel}_i\text{)} \\
\\
\frac{P \vdash \Gamma, x : \langle G_1 \mid \{q\} \rangle \quad Q \vdash \Gamma, x : \langle G_2 \mid \{q\} \rangle \quad q \in \tilde{q}}{x^{qp} \triangleleft (P, Q) \vdash \Gamma, x : \langle (p \rightarrow \tilde{q} : \&\text{in}_1; G_1) \& (p \rightarrow \tilde{q} : \&\text{in}_2; G_2) \mid \{q\} \rangle} \text{ (case)} \\
\\
\frac{P \vdash x_1 : \langle G_1 \mid S_1 \rangle, \dots, x_n : \langle G_n \mid S_n \rangle \quad Q \vdash x_1 : \langle G'_1 \mid S_1 \rangle, \dots, x_n : \langle G'_n \mid S_n \rangle}{P + Q \vdash x_1 : \langle G_1 \oplus G'_1 \mid S_1 \rangle, \dots, x_n : \langle G_n \oplus G'_n \mid S_n \rangle} \text{ (+)} \\
\\
\frac{P \vdash \Gamma_1, x : \langle G_1 \mid S_1 \rangle \quad Q \vdash \Gamma_2, x : \langle G_2 \mid S_2 \rangle \quad S_1 \# S_2 \quad G_3 \simeq_{S_1 \uplus S_2} G_1 \quad S_1 \vee^{S_2} G_2}{P \mid_x Q \vdash \Gamma_1, \Gamma_2, x : \langle G_3 \mid S_1 \uplus S_2 \rangle} \text{ (|)} \\
\\
\frac{}{\text{close}(x) \vdash x : \langle \text{close} \mid \emptyset \rangle} \text{ (close)} \quad \frac{P \vdash \Gamma}{\text{wait}(x).P \vdash \Gamma, x : \langle \text{end} \mid \{p\} \rangle} \text{ (wait)} \\
\\
\frac{P \vdash \Gamma, x : \langle G \mid S \rangle \quad G \downarrow S}{(\nu x)P \vdash \Gamma} \text{ (\nu)} \quad \frac{P \vdash \Gamma \quad \Gamma \simeq \Gamma'}{P \vdash \Gamma'} \text{ (\simeq)} \\
\\
\frac{P \vdash \Gamma, x : \langle G \mid S_1 \rangle \quad S_2 \# \text{fn}(G)}{P \vdash \Gamma, x : \langle G \mid S_1 \cup S_2 \rangle} \text{ (extra)}
\end{array}$$

Fig. 2. Type system for processes.

participants will be able to join that session afterward. To this end, we introduce the notion of *finalized* session type. Intuitively, a type is finalized for a given viewpoint (i.e., a set of participants) if all participants involved in the session are in the viewpoint, there are no occurrence of ω or close (because we need to avoid deadlocks and miscommunications), and that the end of the session is not the end of the process (because we are within a subsession).

Definition 4.3 (Finalized session type). *The judgment $G \downarrow S$, meaning that the session type G is finalized for the set of participants S , is defined as follows:*

$$\begin{array}{c}
\frac{\{p\} \cup \tilde{q} \subseteq S \quad G_1 \downarrow \{p\} \cup \tilde{q} \quad G_2 \downarrow S}{p \rightarrow \tilde{q} : \langle G_1 \rangle; G_2 \downarrow S} \quad \frac{G_1 \downarrow S \quad G_1 \simeq_S G_2}{G_2 \downarrow S} \quad \frac{}{\text{end} \downarrow S} \\
\\
\frac{G_1 \downarrow S \quad G_2 \downarrow S}{G_1 \oplus G_2 \downarrow S} \quad \frac{G_1 \downarrow S \quad G_2 \downarrow S}{G_1 \& G_2 \downarrow S} \quad \frac{\{p\} \cup \tilde{q} \subseteq S \quad G \downarrow S}{p \rightarrow \tilde{q} : \&\text{in}_i; G \downarrow S} \quad \frac{}{0 \downarrow S}
\end{array}$$

Rule (|) is one of the key novelties of this type system. This rule allows us to connect two processes through a shared session *merging* their respective types. The shared session has a merged type, computed by $G_1 \vee^{S_2} G_2$. The definition

of this operator is quite complex and is postponed to Sect. 5. For the time being, it is enough to know that $G_1 \text{ }^{S_1 \vee S_2} G_2$ may not be in \mathfrak{G} , e.g. when G_1, G_2 are not compatible. To guarantee that only valid types are used for the merged session, we have to find some $G_3 \in \mathfrak{G}$ such that $G_3 \simeq_{S_1 \uplus S_2} G_1 \text{ }^{S_1 \vee S_2} G_2$.

The (*extra*) rule allows us to add participants which actually do not interact with the sessions; this is needed for the Subject Reduction.

Remark 4.1. Our rule for parallel composition is similar to a cut rule for linear logic. It may be interesting to compare our rule with the cut rule for linear logic [14], that for binary session types [28], and that for multiparty session types [10]:

$$\frac{\vdash \Gamma, A \quad \vdash \Delta, A^\perp}{\vdash \Gamma, \Delta} \quad \frac{P \vdash \Gamma, x : A \quad Q \vdash \Delta, x : A^\perp}{(\nu x : A)(P \mid Q) \vdash \Gamma, \Delta} \quad \frac{P_i \vdash \Gamma_i, x^{P_i} : A_i \quad G \vDash \{p_i : A_i\}_i}{(\nu x : G)(\Pi_i^x P_i) \vdash \{\Gamma_i\}_i}$$

Each of these rules corresponds to the applications of two rules of our system: the rule (\mid) which merges partial sessions, and the rule (ν) which closes the session. For instance, if we assume that A_1, A_2 , and B are suitable session types, we have the following derivation:

$$\frac{P \vdash \Gamma, x : \langle A_1 \mid S_1 \rangle \quad Q \vdash \Delta, x : \langle A_2 \mid S_2 \rangle \quad A_1 \text{ }^{S_1 \vee S_2} A_2 \simeq_{S_1 \uplus S_2} B}{\frac{P \mid x Q \vdash \Gamma, \Delta, x : \langle B \mid S_1 \uplus S_2 \rangle}{(\nu x)(P \mid x Q) \vdash \Gamma, \Delta} \quad B \downarrow_{S_1 \uplus S_2}}$$

In the case of a multiparty session involving n participants, we would apply (\mid) $n - 1$ times, and then the (ν) rule to close the session. This correspondence (in a logical setting and for binary choreographies) have been previously observed in [9], where the cut rule above is split into two rules (called (Conn) and (Scope)).

5 Merging Partial Session Types

The central part of the type system is the merging algorithm that infers the result of interaction of two partial session types. In this section, we will define the merge function $G_1 \text{ }^{S_1 \vee S_2} G_2$, where G_1, G_2 describe the behavior of a session from the viewpoint of the local participants found in the set S_1 and S_2 , respectively. $G_1 \text{ }^{S_1 \vee S_2} G_2$ then describes the behaviour of the session from the unified viewpoint $S_1 \cup S_2$. In particular, if G_1 and G_2 are intuitively incompatible, then $G_1 \text{ }^{S_1 \vee S_2} G_2$ should contain some occurrence of ω .

To merge two types, we can consider them in DNF; in this way we can recursively reduce the problem to merging chains of communications. Informally, we merge two sequences of communications by considering all possible reorderings which are compatible with each other. This give us a set of all possible merged behaviours, which we glue together using external choices ($\&$). Thus, two types are compatible if they can agree on at least a pair of merged sequences of communications, whatever their internal choices; if no such sequences exist, we get ω as result. Extra complexity is given by the fact that a single communication in the form $p \rightarrow \tilde{q} : \langle G \rangle$ contains a general type; therefore, the function $\text{mcomm}_{S_1, S_2}(C_1, C_2)$ for merging single communications and the function $G_1 \text{ }^{S_1 \vee S_2} G_2$ for merging session types are mutually recursive.

We also need the following helper functions:

- the partial function $\text{cont}_S(G, C)$ takes a chain of communications G and a communication C as input, and returns a type that corresponds to what remains in G after having executed C (up to \simeq_S)
- the decidable predicate $C_1 \overset{S_1}{\heartsuit} \overset{S_2}{S_2} C_2$ tells us whether C_1 and C_2 are mergeable (from their respective viewpoints S_1, S_2)
- the total function $\text{sync}_{S_1, S_2}(G_1, G_2)$ takes two chains of communications G_1 and G_2 as input, and returns all possible tuples (C_1, G'_1, C_2, G') such that $C_1; G'_1 \simeq_{S_1} G_1$, $C_2; G'_2 \simeq_{S_2} G_2$ and C_1 and C_2 are mergeable
- finally, the partial function $\text{map}_{S_1, S_2}(f)(G_1, G_2)$ takes a (partial) function $f : \mathfrak{C} \times \mathfrak{C} \rightarrow \mathfrak{C}$ and two session types in DNF as arguments, and maps f on the pair (G_1, G_2) .

We can then define the partial function $\text{mcomm}_{S_1, S_2}(C_1, C_2)$ and the total function $G_1 \overset{S_1 \vee S_2}{\vee} G_2$. These functions are actually non-deterministic, but $G_1 \overset{S_1 \vee S_2}{\vee} G_2$ is deterministic up to \simeq .

In order to prove termination of these functions, we define the *length* l of session types; besides, we define also the *height* h of communications and session types as the maximal number of nested subsessions. Formally, we have:

$$\begin{aligned}
 l(G_1 \& G_2) &:= l(G_1) + l(G_2) & h(G_1 \& G_2) &:= \max(h(G_1), h(G_2)) \\
 l(G_1 \oplus G_2) &:= l(G_1) + l(G_2) & h(G_1 \oplus G_2) &:= \max(h(G_1), h(G_2)) \\
 l(C; G) &:= 1 + l(G) & h(C; G) &:= \max(h(C), h(G)) \\
 l(G) &:= 1 \quad \text{otherwise.} & h(p \rightarrow \tilde{q} : \langle G \rangle) &:= 1 + h(G) \\
 & & h(C) &:= 0 \quad \text{otherwise.}
 \end{aligned}$$

5.1 Mapping Merging Functions over Session Types

Definition 5.1 (cont). *The partial function $\text{cont}_S(G, C)$ takes as input a chain of communications G and a communication C , and returns some G' in DNF such that $G \simeq_S C; G'$. It is undefined if such G' does not exist.*

Intuitively, $\text{cont}_S(G, C)$ is a kind of Brzozowski derivative that tells us what happens in G after the communication C .

Proposition 5.1. *The function cont is computable, and moreover $l(C; G') = l(G)$ and $h(C; G') = h(G)$.*

Note that $\text{dom}(\text{cont}_S(G, -))$ is finite, and can be computed using Eq. (1) repeatedly.

Definition 5.2 (Function sync). *Let G_1, G_2 be chains of communications in DNF. Let $A_1 = \{(C, G') \mid C \in \text{dom}(\text{cont}_S(G_1, -)), G' = \text{cont}_S(G_1, C)\}$, and $A_2 = \{(C, G') \mid C \in \text{dom}(\text{cont}_S(G_2, -)), G' = \text{cont}_S(G_2, C)\}$. We then define:*

$$\begin{aligned}
 \text{sync}_{S_1, S_2}(f)(G_1, G_2) &:= \{(C_1, C_2, G'_1, G'_2) \mid (C_1, C_2) \neq (1, 1), f(C_1, C_2) \text{ is defined,} \\
 &\quad (C_1, G'_1) \in A_1, (C_2, G'_2) \in A_2\}.
 \end{aligned}$$

Intuitively, $\text{sync}_{S_1, S_2}(f)(G_1, G_2)$ returns a set containing all possible pairs of communications that can be merged, as well as their continuations.

It is important to know whether a communication C_1 (from the viewpoint S_1) and a communication C_2 (from the viewpoint S_2) can correspond to the same communication; in this case, we say that they are *mergeable*. Formally, this notion is defined by the following relation.

Definition 5.3 (Mergeability). *We define $C_1 \overset{S_1 \heartsuit S_2}{\sim} C_2$ as follows:*

$$\frac{\frac{\{p\} \cup \tilde{q}_1 \cup \tilde{q}_2 \subseteq S_1 \cup S_2 \Rightarrow (G_1 \overset{S_1 \vee S_2}{\downarrow} G_2) \downarrow S_1 \cup S_2}{p \in S_1 \Rightarrow \tilde{q}_2 \subseteq \tilde{q}_1 \quad p \in S_2 \Rightarrow \tilde{q}_1 \subseteq \tilde{q}_2 \quad S_1 \cap \tilde{q}_2 \subseteq \tilde{q}_1 \quad S_2 \cap \tilde{q}_1 \subseteq \tilde{q}_2}}{p \rightarrow \tilde{q}_1 : \langle G_1 \rangle \overset{S_1 \heartsuit S_2}{\sim} p \rightarrow \tilde{q}_2 : \langle G_2 \rangle}}{\frac{p \in S_1 \Rightarrow \tilde{q}_2 \subseteq \tilde{q}_1 \quad p \in S_2 \Rightarrow \tilde{q}_1 \subseteq \tilde{q}_2 \quad S_1 \cap \tilde{q}_2 \subseteq \tilde{q}_1 \quad S_2 \cap \tilde{q}_1 \subseteq \tilde{q}_2}{p \rightarrow \tilde{q}_1 : \&\text{in}_i \overset{S_1 \heartsuit S_2}{\sim} p \rightarrow \tilde{q}_2 : \&\text{in}_i} \quad \frac{}{1 \overset{S_1 \heartsuit S_2}{\sim} 1}}{\frac{C_2 \overset{S_2 \heartsuit S_1}{\sim} C_1 \quad \frac{(\{p\} \cup \tilde{q}) \# S_1}{1 \overset{S_1 \heartsuit S_2}{\sim} p \rightarrow \tilde{q} : m} \quad \frac{}{\text{close} \overset{S_1 \heartsuit S_2}{\sim} \text{close}} \quad \frac{}{\text{close} \overset{S_1 \heartsuit S_2}{\sim} \text{end}}}{C_1 \overset{S_1 \heartsuit S_2}{\sim} C_2} \quad \frac{}{1 \overset{S_1 \heartsuit S_2}{\sim} p \rightarrow \tilde{q} : m} \quad \frac{}{\text{close} \overset{S_1 \heartsuit S_2}{\sim} \text{close}} \quad \frac{}{\text{close} \overset{S_1 \heartsuit S_2}{\sim} \text{end}}}}$$

The first rule deserves some explanations. In the first hypothesis, G_1 and G_2 describe sessions whose participants can be only in $\{p\} \cup \tilde{q}_1 \cup \tilde{q}_2$; if all these participants are in $S_1 \cup S_2$, then after the merge all the participants are present and therefore the communication must be safe, because no other participant may join later. This means that, in this case, we have to check that the merge of G_1 and G_2 is finalized. The second hypothesis (and dually the third one) corresponds to the fact that in the (*send*) rule of Fig. 2, the sender specifies all receiving participants, while in (*recv*) a receiver may not know about other receivers; therefore, if $p \rightarrow \tilde{q}_1 : \langle G_1 \rangle$ describes the communication from the point of view of the sender (i.e., $p \in S_1$), then \tilde{q}_2 is a set of receivers only, and must be contained in \tilde{q}_1 . The fourth (and dually the fifth) hypothesis means that if a participant which is known to a process (i.e., in S_1) appears as receiver for other process (i.e., in \tilde{q}_2), then it must appear as a received also by the first process.

Proposition 5.2. $C_1 \overset{S_1 \heartsuit S_2}{\sim} C_2$ is decidable.

Definition 5.4 (Function map). *Let S_1, S_2 be two sets of participants, two types $G_1, G_2 \in \mathfrak{C}$ and a (partial) function $f : \mathfrak{C} \times \mathfrak{C} \rightarrow \mathfrak{C}$ such that*

- G_1 and G_2 are in DNF
- for any C_1, C_2 , we have that $f(C_1, C_2)$ is a terminal communication iff it is defined and both C_1 and C_2 are terminal
- if $f(C_1, C_2)$ is defined, then either both C_1 and C_2 are terminal, or none of them are.

Then, $\text{map}_{S_1, S_2}(f)(G_1, G_2)$ is defined recursively over G_1, G_2 as follows:

- First cases:

$$\begin{aligned} \text{map}_{S_1, S_2}(f)(G_1 \oplus G_2, G_3) &:= \text{map}_{S_1, S_2}(f)(G_1, G_3) \oplus \text{map}_{S_1, S_2}(f)(G_2, G_3) \\ \text{map}_{S_1, S_2}(f)(G_1, G_2 \oplus G_3) &:= \text{map}_{S_1, S_2}(f)(G_1, G_2) \oplus \text{map}_{S_1, S_2}(f)(G_1, G_3) \end{aligned}$$

– If neither of the cases above apply, then we have:

$$\begin{aligned} \text{map}_{S_1, S_2}(f)(G_1 \& G_2, G_3) &:= \text{map}_{S_1, S_2}(f)(G_1, G_3) \& \text{map}_{S_1, S_2}(f)(G_2, G_3) \\ \text{map}_{S_1, S_2}(f)(G_1, G_2 \& G_3) &:= \text{map}_{S_1, S_2}(f)(G_1, G_2) \& \text{map}_{S_1, S_2}(f)(G_1, G_3) \end{aligned}$$

– If G_1, G_2 are both chains of communications and at least one of them is not a terminal communication, we pose $B := \text{sync}_{S_1, S_2}(f)(G_1, G_2)$ and we have:

- If G_1 or G_2 ends with 0, $\text{map}_{S_1, S_2}(f)(G_1, G_2) := 0$.
- If G_1 or G_2 ends with ω , or if $B = \emptyset$, then $\text{map}_{S_1, S_2}(f)(G_1, G_2) := \omega$.
- Otherwise:

$$\text{map}_{S_1, S_2}(f)(G_1, G_2) := \&\{f(C_1, C_2); \text{map}_{S_1, S_2}(f)(G'_1, G'_2) \mid (C_1, C_2, G'_1, G'_2) \in B\}$$

– If G_1 and G_2 are both terminal communications, then:

$$\text{map}_{S_1, S_2}(f)(G_1, G_2) := \begin{cases} 0 & \text{if } G_1 \text{ or } G_2 \text{ is } 0 \\ f(G_1, G_2) & \text{if } f(G_1, G_2) \text{ is defined} \\ \omega & \text{otherwise.} \end{cases}$$

The two conditions on f guarantee that $\text{map}_{S_1, S_2}(f)(G_1, G_2)$ is well-defined in the last two cases, when f is applied to G_1, G_2 or to the chains C_1, C_2 .

Proposition 5.3. *Termination of map is ensured by induction on $l(G_1) + l(G_2)$.*

Note that, when we computing $\text{map}_{S_1, S_2}(f)(G_1, G_2)$, every application of f is of the form $f_{S_1, S_2}(C_1, C_2)$, where $h(C_1) + h(C_2) \leq h(G_1) + h(G_2)$.

5.2 Merging Communications and Session Types

We now define the partial function $\text{mcomm}_{S_1, S_2}(C_1, C_2)$ which merges compatible communications C_1 (from the viewpoint S_1) and C_2 (from the viewpoint S_2) and returns, if possible, the new communication from the merged viewpoints $S_1 \cup S_2$. We also define by mutual recursion the merging function for session types, which is just a shorthand for map applied to mcomm :

$$G_1 \text{ }^{S_1 \vee S_2} G_2 := \text{map}_{S_1, S_2}(\text{mcomm}_{S_1, S_2})(G_1, G_2)$$

We suppose that G_1 and G_2 are in DNFs, but it can be applied to any session types by rewriting them in DNF.

Definition 5.5 (Function mcomm). *If $C_1 \text{ }^{S_1 \heartsuit S_2} C_2$, then:*

$$\begin{aligned} \text{mcomm}_{S_1, S_2}(p \rightarrow \tilde{q} : \&\text{in}_i, p \rightarrow \tilde{q}' : \&\text{in}_i) &:= p \rightarrow (\tilde{q} \cup \tilde{q}') : \&\text{in}_i \\ \text{mcomm}_{S_1, S_2}(p \rightarrow \tilde{q} : \langle G_1 \rangle, p \rightarrow \tilde{q}' : \langle G_2 \rangle) &:= p \rightarrow (\tilde{q} \cup \tilde{q}') : \langle G_1 \text{ }^{S_1 \vee S_2} G_2 \rangle \\ \text{mcomm}_{S_1, S_2}(1, C) &:= C \\ \text{mcomm}_{S_1, S_2}(C, 1) &:= C \\ \text{mcomm}_{S_1, S_2}(C, \text{close}) &:= C \\ \text{mcomm}_{S_1, S_2}(\text{close}, C) &:= C \end{aligned}$$

Otherwise, $\text{mcomm}_{S_1, S_2}(C_1, C_2)$ is undefined.

Proposition 5.4. *Termination is ensured by induction on $h(C_1) + h(C_2)$.*

Example 5.1. Continuing Example 3.1, let us recall the types of participants p, r :

$$\begin{aligned} G_p &:= G'_p \oplus G''_p & G_r &:= G'_r \& G''_r \\ G'_p &:= p \rightarrow q : \&in_1; p \rightarrow r : \langle \text{end} \rangle; \text{close} & G''_p &:= p \rightarrow q : \&in_2; \text{close} \\ G'_r &:= q \rightarrow r : \&in_1; p \rightarrow r : \langle \text{close} \rangle; \text{close} & G''_r &:= q \rightarrow r : \&in_2; \text{close} \end{aligned}$$

We have that:

$$\begin{aligned} \text{dom}(\text{cont}_{\{p\}}(G'_p)) &= p \rightarrow q : \&in_1 & \text{dom}(\text{cont}_{\{p\}}(G''_p)) &= p \rightarrow q : \&in_2 \\ \text{dom}(\text{cont}_{\{r\}}(G'_r)) &= q \rightarrow r : \&in_1 & \text{dom}(\text{cont}_{\{r\}}(G''_r)) &= q \rightarrow r : \&in_2 \end{aligned}$$

As a consequence, we have for instance that: $\text{sync}_{\{p\},\{q\}}(G_1, G'_1) = \{(p \rightarrow q : \&in_1, 1, (p \rightarrow r : \langle \text{end} \rangle; \text{close}), G'_1), (1, q \rightarrow r : \&in_1, G_1, (p \rightarrow r : \langle \text{close} \rangle; \text{close}))\}$.

We have that:

$$\begin{aligned} G'_p \{p\} \vee \{r\} G'_r &= (p \rightarrow q : \&in_1; q \rightarrow r : \&in_1; p \rightarrow r : \langle \text{end} \rangle; \text{close}) \& \\ &\quad (q \rightarrow r : \&in_1; p \rightarrow q : \&in_1; p \rightarrow r : \langle \text{end} \rangle; \text{close}) \\ G'_p \{p\} \vee \{r\} G''_r &= (p \rightarrow q : \&in_1; q \rightarrow r : \&in_2; \omega) \& (q \rightarrow r : \&in_2; p \rightarrow q : \&in_1; \omega) \\ G''_p \{p\} \vee \{r\} G'_r &= (p \rightarrow q : \&in_2; q \rightarrow r : \&in_1; \omega) \& (q \rightarrow r : \&in_1; p \rightarrow q : \&in_2; \omega) \\ G''_p \{p\} \vee \{r\} G''_r &= (p \rightarrow q : \&in_2; q \rightarrow r : \&in_2; \text{close}) \& \\ &\quad (q \rightarrow r : \&in_2; p \rightarrow q : \&in_2; \text{close}) \end{aligned}$$

and finally

$$\begin{aligned} G_p \{p\} \vee \{r\} G_r &= ((G'_p \{p\} \vee \{r\} G'_r) \& (G'_p \{p\} \vee \{r\} G''_r)) \oplus \\ &\quad ((G''_p \{p\} \vee \{r\} G'_r) \& (G''_p \{p\} \vee \{r\} G''_r)) \\ &\simeq_{\{p,r\}} (p \rightarrow q : \&in_1; q \rightarrow r : \&in_1; p \rightarrow r : \langle \text{end} \rangle; \text{close}) \oplus \\ &\quad (p \rightarrow q : \&in_2; q \rightarrow r : \&in_2; \text{close}) \end{aligned}$$

6 Subject Reduction and Progress

In this section we state two main properties of session types, *subject reduction* and *progress*, which guarantee that “well-typed systems cannot go wrong”. To this end, we first define a reduction semantics for partial session types.

Definition 6.1 (Reductions for session types). *Actions γ for session types are defined as*

$$\gamma ::= + \mid p \rightarrow \tilde{q} : \langle \cdot \rangle \mid p \rightarrow \tilde{q} : \&in_i$$

We write $G_1 \xrightarrow{\gamma}_S G_2$ for a transition from G_1 to G_2 from the viewpoint of S under the action γ . This relation is defined as follows:

$$\begin{aligned} G_1 \oplus G_2 &\xrightarrow{+}_S G_i & p \rightarrow \tilde{q} : \langle G_1 \rangle; G_2 &\xrightarrow{p \rightarrow \tilde{q} : \langle \cdot \rangle}_S G_2 \\ p \rightarrow \tilde{q} : \&in_i; G &\xrightarrow{p \rightarrow \tilde{q} : \&in_i}_S G & \frac{G_1 \xrightarrow{\gamma}_S G' \quad G_1 \simeq_S G_2}{G_2 \xrightarrow{\gamma}_S G'} \end{aligned}$$

Note that transitions are not deterministic, in particular $G \simeq_S G \oplus G$, therefore we always have $G \xrightarrow{+} G$, which is useful in case we are reducing an internal choice which is irrelevant for G .

Definition 6.2 (Reduction for environments). *Reductions for environments are labelled by actions for processes α , and are defined as follows:*

$$\begin{array}{c} \frac{}{\cdot \xrightarrow{\alpha} \cdot} \quad \frac{}{\Gamma \xrightarrow{\tau} \Gamma} \quad \frac{G_1 \xrightarrow{+}_S G_2 \quad \Gamma_1 \xrightarrow{+} \Gamma_2}{x : \langle G_1 \mid S \rangle, \Gamma_1 \xrightarrow{+} x : \langle G_2 \mid S \rangle, \Gamma_2} \\ \\ \frac{G_1 \xrightarrow{\gamma}_S G_2}{x : \langle G_1 \mid S \rangle, \Gamma \xrightarrow{x:\gamma} x : \langle G_2 \mid S \rangle, \Gamma} \quad \frac{\Gamma_1 \xrightarrow{y:\gamma} \Gamma_2 \quad x \neq y}{x : \langle G \mid S \rangle, \Gamma_1 \xrightarrow{y:\gamma} x : \langle G \mid S \rangle, \Gamma_2} \end{array}$$

The type system enjoys the following properties:

Theorem 6.1 (Subject equivalence). *If $P \vdash \Gamma$ and $P \equiv Q$, then $Q \vdash \Gamma$.*

From now on, we can consider processes equal modulo \equiv .

Theorem 6.2 (Subject reduction). *If $P_1 \vdash \Gamma_1$ and $P_1 \xrightarrow{\alpha} P_2$, then for some Γ_2 , we have $P_2 \vdash \Gamma_2$ and $\Gamma_1 \xrightarrow{\alpha} \Gamma_2$.*

Remark 6.1. In earlier work about MPST, usually subject reduction requires some *consistency* condition over the typing environment Γ (see, e.g., [26]). In our development, this condition is not explicitly needed because the type rules for processes ensure that environments are consistent; hence, the derivability of $P_1 \vdash \Gamma_1$ implies that no session in Γ_1 has the type ω .

Progress. In usual session types, the progress property means that well-typed systems can always proceed, and in particular they are deadlock-free. In our case, well-typed systems can still contain processes which cannot proceed not due to a deadlock or miscommunication but due to some missing participant.

Example 6.1. Let us consider $P = \bar{x}^{pq} \triangleright \&\text{in}_1.\text{close}(x)$. This process is typable ($P \vdash x : \langle p \rightarrow q : \&\text{in}_1; \text{close} \mid \{p\} \rangle$), yet it is stuck. It can be completed into a redex $P \mid_x Q$, with $Q = x^{qp} \triangleleft (Q_1, Q_2)$. In fact, P can be seen as the *restriction* of $P \mid_x Q$ on session x with participants in $\{p\}$. Hence, P is preempted by x and so it can be considered a correct process, waiting for the missing participant.

Therefore, in order to define the progress property for our system, we need to define the restriction of a process to a given set of local participants.

Definition 6.3 (Restriction). *We define the restriction of a term P on session x with participants in S (noted $P \downarrow_S x$) as follows:*

$$\begin{array}{ll} \bar{x}^{p\bar{q}}(y).P \downarrow_S x = \text{close}(x) \text{ if } p \notin S & x^{pq}(y).(P \parallel Q) \downarrow_S x = \text{close}(x) \text{ if } p \notin S \\ \bar{x}^{p\bar{q}} \triangleright \text{in}_i.P \downarrow_S x = \text{close}(x) \text{ if } p \notin S & x^{pq} \triangleleft (P, Q) \downarrow_S x = \text{close}(x) \text{ if } p \notin S \\ P \mid_x Q \downarrow_S x = (P \downarrow_S x) \mid_x (Q \downarrow_S x) & P \downarrow_S x = P \text{ otherwise} \end{array}$$

Definition 6.4 (Preemption). We say that a session x with type $G \in \mathfrak{G}$ and local participants S preempts P (noted $x : \langle G \mid S \rangle \gg_g P$) when one of these condition occurs:

- $x : \langle p \rightarrow \tilde{q} : \langle G_1 \rangle; G_2 \mid S \rangle \gg_g ((\bar{x}^{p\tilde{q}}(y).R \mid_x \Pi_i^x(x^{q_i p}(y).(P_i \parallel Q_i))) \downarrow_S x) \mid_x P$ if $G_2 \simeq_S C$ where C is terminal, or $x : \langle G_2 \mid S - \{p, \tilde{q}\} \rangle \gg_g P$
- $x : \langle p \rightarrow \tilde{q} : \&\text{in}_i; G \mid S \rangle \gg_g (\bar{x}^{p\tilde{q}} \triangleright \text{in}_i.R \mid_x \Pi_j^x x^{q_j p} \triangleleft (P_{1,j}, P_{2,j}) \downarrow_S x) \mid_x P$ if $G_2 \simeq_S C$ where C is terminal, or $x : \langle G \mid S - \{p, \tilde{q}\} \rangle \gg_g P$
- $x : \langle \text{close} \mid S \rangle \gg_g \text{close}(x)$
- $x : \langle \text{end} \mid S \rangle \gg_g \text{wait}(x).P$
- $x : \langle G_1 \oplus G_2 \mid S \rangle \gg_g U$ if $x : \langle G_1 \mid S \rangle \gg_g P$ or $x : \langle G_2 \mid S \rangle \gg_g P$
- $x : \langle G_1 \& G_2 \mid S \rangle \gg_g P$ if $x : \langle G_1 \mid S \rangle \gg_g P$ and $x : \langle G_2 \mid S \rangle \gg_g P$
- $x : \langle G \mid S \rangle \gg_g P$ if $x : \langle G \mid S \rangle \gg_g P'$ and $P \equiv P'$

Definition 6.5 (Contextual preemption). We define $x : \langle G \mid S \rangle \gg_c P$ for some $\mathcal{C}[_]$, P' , we have that $P \equiv \mathcal{C}[P']$, $x \notin \text{fn}(\mathcal{C}[_])$, and $x : \langle G \mid S \rangle \gg_g P'$.

Intuitively, $x : \langle G \mid S \rangle \gg_c P$ means that every local participant in S is ready to trigger its respective communication described in G . As a consequence, there is no deadlock for x : if all the concerned participants are present there is a redex, otherwise we are blocked due to the absence of some sender or receiver. The following lemma states that if a session is finalized and preempted, then the process (with the session restricted) contains a redex.

Lemma 6.1. 1. If $G \downarrow S$ and $x : \langle G \mid S \rangle \gg_g P$, then $(\nu x)P$ has a redex.
 2. If $G \downarrow S$ and $x : \langle G \mid S \rangle \gg_c P$, then $(\nu x)P$ has a redex.

Theorem 6.3 (Progress). If $P \vdash \Gamma$ then there is a redex in P , or for some $x : \langle G \mid S \rangle \in \Gamma$ we have $x : \langle G \mid S \rangle \gg_c P$.

7 Related Work

The problem of composing session types has been faced in several related work. Compositional choreographies are discussed in [23], with the same motivations as ours, but from a different perspective. The authors manage to compose choreographies using global types, but the global type of shared channels has to be the same. This is in contrast with our approach, where the processes may have different session types that we merge during the composition. Moreover, also their typing judgments use sets of participants (there called *roles*); more precisely, the types for channels keep track of the “active” role, the set of all roles in the global type, and the roles actually implemented by the choreography under typing. On the other hand, we do not need to specify neither the complete set of participants nor the “active” role, in typing sessions.

Synthesis of choreography from local types has been studied also in [21], but with no notion of “partial types” and no distinction between internal/external choice. Graphical representations of choreographies (as communicating finite-state machines) and global types have been used in [22], where an algorithm for constructing a global graph from asynchronous interactions is given.

An interesting approach based on *gateways* has been investigated in [2–5]: two independent global types G_1 and G_2 with different participants can be composed through participant h in G_1 and k in G_2 where h and k relay the message they receive to each other. Therefore, in this approach the two session types G_1, G_2 are connected by the gateway but not really merged, as in our approach. Finally, [26] do not use global types altogether: behaviours of systems are represented by sets of local types, over which no consistency conditions are required, and behavioural properties can be verified using model checking techniques.

A problem similar to ours is considered in [8], where the authors introduce a type system for the Conversation Calculus, a model for service-oriented computing. Conversation types of parallel processes can be merged like in our approach, but the underlying computational model is quite different.

Semantics of concurrent processes can be given using Mazurkiewicz trace languages [25]. Semantics can also be defined using event structures, as in [11], where they are used for defining equivalent semantics for processes and their global types. Interestingly, the semantics for global types proposed in [11] is similar to the representation of Mazurkiewicz trace languages as event structures given in [25]. Mazurkiewicz trace languages have been also used to characterize testing preorders on multiparty scenarios [13]. A denotational semantics based on Brzozowski derivatives that corresponds to bisimilarity is given in [20].

Another semantics of processes (but for binary session types) that records exchanged informations is given in [1]. This semantics is similar to the relation-based model of linear logic [6], and not based on traces. It would be interesting to investigate if this alternative semantics can be extended to MPST and to interpret the merge operation. The relationship between category theory and session types has been investigated also in [19, 27].

8 Conclusions

In this paper, we have introduced *partial sessions* and *partial (multiparty) session types*, extending global session types with the possibility to type also sessions with missing participants. Sessions with the same name but observed by different participants can be merged if their types are *compatible*; in this case, the type for the unified session can be derived compositionally from the types of components. To this end we have provided a merging algorithm, which allows us to detect incompatible types, due to miscommunications or deadlocks, as early as possible; this differs from usual session type systems which delay all the checks to when the system is completed (i.e., at the restriction rule). Therefore, in this theory the distinction between local and global types vanishes. We have also generalised the notion of *progress* to accommodate the case when a partial session cannot progress not due to a deadlock, but to some missing participant.

Future Work. An interesting application of partial session types would be in the verification of composition of components, like e.g. containers *a la* Docker; to this end, we can think of defining a typing discipline similar to the one presented in this paper, but tailored for a formal models of containers, like that in [7].

We claim that for the type system presented in this paper both type checking and type inference are decidable. The idea is that, in order to be typable, the structure of a process has to match the structure of the type(s), up-to type equivalence; hence, the typing derivation is bounded by the complexity of process terms. At worst, this bound is exponential, as in the application of type equivalence rule we have to explore a possibly exponential space of equivalent types; however, this limit could be improved by some algorithmic machinery concerning the normal form of types, which we leave to future work.

The current merging algorithm returns types that may contain many equivalent subterms; a future work could be to define shorter and more efficient representations. Another interesting aspect of this algorithm is that it is defined by two functions (`map` and `mcomm`), which can be updated separately in future variations; in particular, adding recursion only requires to update the function `map`, while adding new kinds of communication, or changing how communications are merged, only requires to update the function `mcomm`. The correctness of this algorithm can be proved with respect to a categorical semantics for session types based on traces, which we leave to the extended version of this work.

In this paper we have considered a calculus with synchronous multicast, along the lines of [10, 28] and others. However, it would be interesting to extend the definitions and results of this paper to an *asynchronous* version of the calculus. This is not immediate, as it requires non-trivial changes in the typing systems and especially in the (already quite complex) merging operation.

Following the Liskov substitution principle, we could define a subtyping relation by seeing $\&$ and \oplus as the meet and join operator of a lattice, respectively. However, a semantical understanding of this subtyping relation is not clear yet.

One intriguing possible extension would be to add some form of *encapsulation*. For instance, if we have the type $p \rightarrow q : m_1; q \rightarrow r : m_2; p \rightarrow r : m_3; \text{close}$; close from the viewpoint of $\{q, r\}$, then we could be tempted to “erase” the communication $q \rightarrow r : m_2$, since this communication is purely internal, but this erasure would not be compatible with equivalence:

$$p \rightarrow q : m_1; q \rightarrow r : m_2; p \rightarrow r : m_3; \text{close} \not\approx_{\{q,r\}} p \rightarrow q : m_3; q \rightarrow r : m_2; \\ p \rightarrow r : m_1; \text{close}$$

but $p \rightarrow q : m_1; p \rightarrow r : m_3; \text{close} \simeq_{\{q,r\}} p \rightarrow q : m_3; p \rightarrow r : m_1; \text{close}$

How to add a form of encapsulation to our type system is an open question.

Finally, to guarantee the correctness of most complex proofs and definitions of this paper, it would be useful to formalise them in a proof assistant, like Coq.

Acknowledgments. We are grateful to Mariangiola Dezani-Ciancaglini, Marco Peressotti and the anonymous reviewers for their useful remarks on the preliminar version of this paper.







References

1. Atkey, R.: Observed communication semantics for classical processes. In: Yang, H. (ed.) ESOP 2017. LNCS, vol. 10201, pp. 56–82. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54434-1_3
2. Barbanera, F., Dezani-Ciancaglini, M.: Open multiparty sessions. In: Proceedings of the ICE. EPTCS, vol. 304, pp. 77–96 (2019)
3. Barbanera, F., Dezani-Ciancaglini, M., Lanese, I., Tuosto, E.: Composition and decomposition of multiparty sessions. *J. Log. Algebraic Methods Program.* **119**, 100620 (2021)
4. Barbanera, F., Lanese, I., Tuosto, E.: Composing communicating systems, synchronously. In: Margaria, T., Steffen, B. (eds.) ISO/LA 2020. LNCS, vol. 12476, pp. 39–59. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_3
5. Barbanera, F., de’ Liguoro, U., Hennicker, R.: Global types for open systems. In: Proceedings of ICE. EPTCS, vol. 279, pp. 4–20 (2018)
6. Barr, M.: *-autonomous categories and linear logic. *Math. Struct. Comput. Sci.* **1**(2), 159–178 (1991). <https://doi.org/10.1017/S0960129500001274>
7. Burco, F., Miculan, M., Peressotti, M.: Towards a formal model for composable container systems. In: Hung, C., Cerný, T., Shin, D., Bechini, A. (eds.) SAC 2020: The 35th ACM/SIGAPP Symposium on Applied Computing, pp. 173–175. ACM (2020). <https://doi.org/10.1145/3341105.3374121>
8. Caires, L., Vieira, H.T.: Conversation types. *Theoret. Comput. Sci.* **411**(51–52), 4399–4440 (2010)
9. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. *Distrib. Comput.* **31**(1), 51–67 (2018)
10. Carbone, M., Montesi, F., Schürmann, C., Yoshida, N.: Multiparty session types as coherence proofs. *Acta Informatica* **54**(3), 243–269 (2017)
11. Castellani, I., Dezani-Ciancaglini, M., Giannini, P.: Event structure semantics for multiparty sessions. In: Boreale, M., Corradini, F., Loreti, M., Pugliese, R. (eds.) Models, Languages, and Tools for Concurrent and Distributed Programming. LNCS, vol. 11665, pp. 340–363. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21485-2_19
12. 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)
13. De Nicola, R., Melgratti, H.: Multiparty testing preorders. In: Ganty, P., Loreti, M. (eds.) TGC 2015. LNCS, vol. 9533, pp. 16–31. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-28766-9_2
14. Girard, J.Y.: Linear logic. *Theoret. Comput. Sci.* **50**(1), 1–101 (1987)
15. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). <https://doi.org/10.1007/BFb0053567>
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the POPL 2008, pp. 273–284. ACM (2008). <https://doi.org/10.1145/1328438.1328472>
17. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. *J. ACM* **63**(1), 9:1–9:67 (2016). <https://doi.org/10.1145/2827695>
18. Jespersen, T.B.L., Munksgaard, P., Larsen, K.F.: Session types for rust. In: Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, pp. 13–22 (2015)

19. Keizer, A.C., Basold, H., Pérez, J.A.: Session coalgebras: a coalgebraic view on session types and communication protocols. In: ESOP 2021. LNCS, vol. 12648, pp. 375–403. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72019-3_14
20. Kokke, W., Montesi, F., Peressotti, M.: Better late than never: a fully-abstract semantics for classical processes. *Proc. ACM Program. Lang.* **3**(POPL) (2019). <https://doi.org/10.1145/3290337>
21. Lange, J., Tuosto, E.: Synthesising choreographies from local session types. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 225–239. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32940-1_17
22. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. In: POPL, pp. 221–232. ACM (2015)
23. Montesi, F., Yoshida, N.: Compositional choreographies. In: D’Argenio, P.R., Melgratti, 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. Neubauer, M., Thiemann, P.: An implementation of session types. In: Jayaraman, B. (ed.) PADL 2004. LNCS, vol. 3057, pp. 56–70. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24836-1_5
25. Nielsen, M.: Models for concurrency. In: Tarlecki, A. (ed.) MFCS 1991. LNCS, vol. 520, pp. 43–46. Springer, Heidelberg (1991). https://doi.org/10.1007/3-540-54345-7_47
26. Scalas, A., Yoshida, N.: Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.* **3**(POPL), 1–29 (2019)
27. Toninho, B., Yoshida, N.: Polymorphic session processes as morphisms. In: Alvim, M.S., Chatzikokolakis, K., Olarte, C., Valencia, F. (eds.) *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy*. LNCS, vol. 11760, pp. 101–117. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-31175-9_7
28. Wadler, P.: Propositions as sessions. *J. Funct. Program.* **24**(2–3), 384–418 (2014)



A Canonical Algebra of Open Transition Systems

Elena Di Lavore¹ , Alessandro Gianola² , Mario Román¹  ,
Nicoletta Sabadini³ , and Paweł Sobociński¹ 

¹ Tallinn University of Technology, Ehitajate tee 5, 12616 Tallinn, Estonia
elendi@ttu.ee

² Free University of Bozen-Bolzano, Piazza Domenicani, 3, 39100 Bolzano, BZ, Italy

³ Università degli Studi dell'Insubria, Via Ravasi, 2, 21100 Varese, VA, Italy
nicoletta.sabadini@uninsubria.it

Abstract. Feedback and state are closely interrelated concepts. Categories with feedback, originally proposed by Katis, Sabadini and Walters, are a weakening of the notion of traced monoidal categories, with several pertinent applications in computer science. The construction of the *free* such categories has appeared in several different contexts, and can be considered as *state bootstrapping*. We show that a categorical algebra for *open transition systems*, $\mathbf{Span}(\mathbf{Graph})_*$, also due to Katis, Sabadini and Walters, is the free category with feedback over $\mathbf{Span}(\mathbf{Set})$. This algebra of transition systems is obtained by adding state to an algebra of predicates, and therefore $\mathbf{Span}(\mathbf{Graph})_*$ is the canonical such algebra.

1 Introduction

1.1 State from Feedback

A remarkable fact from electronic circuit design is how data-storing components can be built out of a combination of *stateless components* and *feedback*. A famous example is the (set-reset) “NOR latch”: a circuit with two stable configurations that *stores* one bit.

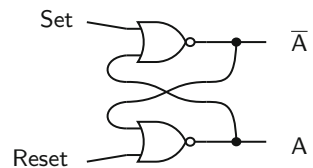


Fig. 1. NOR latch.

The NOR latch is controlled by two inputs, Set and Reset. Activating the first sets the output value to $A = 1$; activating the second makes the output value return to $A = 0$. This change is permanent: even when both Set and Reset are deactivated, the feedback loop maintains the last value the circuit was set to¹—to wit, a bit of data has

¹ In its original description: “the relay is designed to produce a large and **permanent** change in the current flowing in an electrical circuit by means of a small electrical stimulus received from the outside” ([12], emphasis added).

Di Lavore, Román and Sobociński were supported by the European Union through the ESF funded Estonian IT Academy research measure (2014-2020.4.05.19-0001). This work was also supported by the Estonian Research Council grant PRG1210.

been conjured out of thin air. In this paper we exhibit this as an instance of an abstract phenomenon: the universal way of adding feedback to a theory of processes consists of endowing each process with a *state space*.

Indeed, there is a natural weakening of the notion of traced monoidal categories called *categories with feedback* [30]. The construction of the *free* category with feedback coincides with a “state-bootstrapping” construction, $\text{St}(\bullet)$, that appears in several different contexts in the literature [7, 23, 26]. We recall this construction and its mathematical status (Theorem 2.5), which can be summed up by the following intuition:

$$\text{Theory of Processes} + \text{Feedback} = \text{Theory of Stateful Processes.}$$

1.2 The Algebra of Transition Systems

Our primary focus is the $\text{Span}(\mathbf{Graph})$ model of concurrency, introduced in [27] as a categorical algebra of *communicating state machines*, or—equivalently—*open transition systems*. Open transition systems interact by synchronization, producing a *simultaneous change of state*. This corresponds to a composition of spans, realized by taking a pullback in \mathbf{Graph} . The dual algebra of $\text{Cospan}(\mathbf{Graph})$ was introduced in [29]. It complements $\text{Span}(\mathbf{Graph})$ by adding the operation of *communicating-sequential composition* [17].

Informally, a morphism of $\text{Span}(\mathbf{Graph})$ is a state machine with states and transitions, i.e. a finite graph given by the ‘head’ of the span. The transition system is equipped with interfaces or *communication ports*, and every transition is labeled by the effect it produces in *all* its interfaces. We give examples below.

1.3 Stateful and Stateless Components

In Fig. 2, we depict two open transition systems as arrows of $\text{Span}(\mathbf{Graph})$. The first represents a NOR gate $\mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$. The diagram below left (Fig. 2) is a graphical rendering of the corresponding span $\mathbb{B} \times \mathbb{B} \leftarrow N \rightarrow \mathbb{B}$, where \mathbb{B} is a single-vertex graph with two edges, corresponding to the signals $\{0, 1\}$, the *unlabeled* graph depicted within the bubble is N , and the labels witness the action of two homomorphisms, respectively $N \rightarrow \mathbb{B} \times \mathbb{B}$ and $N \rightarrow \mathbb{B}$. Here each transition represents one of the valid input/output configurations of the gate. NOR gates are *stateless* components; the graph N has a single vertex.

The second component is a span $L = \{\text{Set}, \text{Reset}, \text{Idle}\} \rightarrow \{\text{A}, \bar{\text{A}}\} = R$ that models a set-reset latch. The diagram below right (Fig. 2), again, is a convenient illustration of the span $L \leftarrow D \rightarrow R$. Latches store one bit of information, they are *stateful components*; consequently, their transition graph has two states.

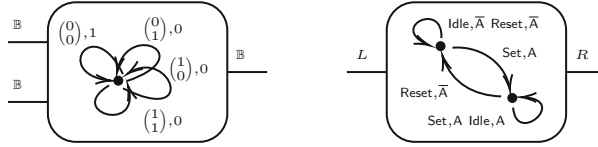


Fig. 2. A NOR gate and set-reset latch, in $\text{Span}(\text{Graph})$.

In both cases, the interfaces on $\text{Span}/\text{Cospan}(\text{Graph})$ are stateless: indeed, they are determined by a mere set – the self-loops of a single-vertex graph. This is a restriction that occurs rather frequently: the important subcategory of $\text{Span}(\text{Graph})$, the one that we can clearly conceptually explain as *transition systems with interfaces*, is the full subcategory of $\text{Span}(\text{Graph})$ restricted to objects that are single-vertex graphs, which we denote by $\text{Span}(\text{Graph})_*$. Analogously, the relevant subcategory of $\text{Cospan}(\text{Graph})$ is $\text{Cospan}(\text{Graph})_*$, the full subcategory on sets, or graphs with an empty set of edges.

Definition. $\text{Span}(\text{Graph})_*$ is the full subcategory of $\text{Span}(\text{Graph})$ with objects the single-vertex graphs.

The problem with $\text{Span}(\text{Graph})_*$ is that it is mysterious from the categorical point of view; the morphisms are graphs, but the boundaries are sets. *Decorated* and *structured* spans and cospans [3, 14] are frameworks that capture such phenomena, which occur frequently when composing network structures. Nevertheless, they do not answer the question of *why* they arise naturally.

1.4 Canonicity and Our Original Contribution

Universal constructions, such as “state-bootstrapping” $\text{St}(\bullet)$, characterize the object of interest up to equivalence, making it *the canonical object* satisfying some properties. This is the key to side-stepping Abramsky’s concern [1]: because of the lack of consensus about the intrinsic primitives of concurrency, we risk making our results too dependent on a specific syntax. It is thus important to characterize existing models of concurrency in terms of universal properties.

The main contribution of this paper is the characterization of $\text{Span}(\text{Graph})_*$ in terms of a universal property: it is equivalent to the free category with feedback over the category of spans of functions. We now state this more formally:

Theorem. *The free category with feedback over $\text{Span}(\text{Set})$ is isomorphic to $\text{Span}(\text{Graph})_*$, the full subcategory of $\text{Span}(\text{Graph})$ given by single-vertex graphs. That is, there is an isomorphism of categories*

$$\text{St}(\text{Span}(\text{Set})) \cong \text{Span}(\text{Graph})_*.$$

Given that $\text{Span}(\text{Set})$, the category of spans of functions, can be considered an *algebra of predicates* [4, 10], the high level intuition that summarizes our main contribution (Theorem 3.8) can be stated as:

Algebra of Predicates + Feedback = Algebra of Transition Systems.

We similarly prove (in Sect. 3.4) that the free [category with feedback](#) over $\mathbf{Cospan}(\mathbf{Set})$ is isomorphic to $\mathbf{Cospan}(\mathbf{Graph})_*$, the full subcategory on discrete graphs of $\mathbf{Cospan}(\mathbf{Graph})$.

1.5 Related Work

$\mathbf{Span}/\mathbf{Cospan}(\mathbf{Graph})$ has been extensively used for the modeling of concurrent systems [9, 15–17, 27, 29, 38, 41, 42]. Similar approaches to compositional modeling of networks have used *decorated* and *structured cospans* [3, 14]. However, $\mathbf{Span}(\mathbf{Graph})_*$ has not previously been characterized in terms of a universal property.

In [30], the $\mathbf{St}(\bullet)$ construction (under a different name) is exhibited as the free *category with feedback*. Categories with feedback have been arguably underappreciated but, at the same time, the $\mathbf{St}(\bullet)$ construction has made multiple appearances as a “state bootstrapping” technique across the literature. The $\mathbf{St}(\bullet)$ construction is used to describe a string diagrammatic syntax for *concurrency theory* in [7]; a variant of it had been previously applied in the setting of *cartesian bicategories* in [26]; and it was again rediscovered to describe a *memoryful geometry of interaction* in [23]. However, a coherent account of both categories with feedback and their relation with these stateful extensions has not previously appeared. This motivates our extensive preliminaries in Sects. 2.1 and 2.2.

1.6 Synopsis

Section 2 contains preliminary discussions on [traced monoidal categories](#) and [categories with feedback](#); it explicitly describes $\mathbf{St}(\bullet)$, the free [category with feedback](#). It collects mainly expository material. Section 3 exhibits a universal property for the $\mathbf{Span}(\mathbf{Graph})_*$ and $\mathbf{Cospan}(\mathbf{Graph})_*$ models of concurrency and Sect. 3.5 highlights a specific application.

1.7 Conventions

We write composition of morphisms in diagrammatic order, $f;g$. When describing morphisms in a [symmetric monoidal category](#) we omit the associators and unitors, implicitly using the *coherence theorem* [33, Theorem 2.1, Chapter VII].

2 Preliminaries: Categories with Feedback

[Categories with feedback](#) were introduced in [30], and motivated by examples such as *Elgot automata* [13], *iteration theories* [6] and *continuous dynamical systems* [28]. We recall the details below, contrast them with the stronger notion of *traced monoidal categories* in Sect. 2.2, discuss the relationship between feedback and delay in Sect. 2.3, recall the construction of a free category with feedback in Sect. 2.4, and give examples in Sect. 2.5.

2.1 Categories with Feedback

A *feedback operator*, $\text{fbk}(\bullet)$, takes a morphism $S \otimes A \rightarrow S \otimes B$ and “feeds back” one of its outputs to one of its inputs of the same type, yielding a morphism $A \rightarrow B$ (Fig. 3, left). When using string diagrams, we depict the action of the *feedback operator* as a loop with a double arrowtip (Fig. 3, right).

$$\frac{f: S \otimes A \rightarrow S \otimes B}{\text{fbk}_S(f): A \rightarrow B} \qquad \begin{array}{c} \overleftarrow{S} \\ \boxed{f} \\ \overrightarrow{A} \quad \overrightarrow{B} \end{array}$$

Fig. 3. Type and graphical notation for the operator $\text{fbk}_S(\bullet)$.

Capturing a reasonable notion of feedback requires the operator to interact nicely with the flow imposed by the structure of a symmetric monoidal category. This interaction is expressed by a few straightforward axioms.

Definition 2.1. A *category with feedback* [30] is a symmetric monoidal category \mathbf{C} endowed with an operator $\text{fbk}_S: \mathbf{C}(S \otimes A, S \otimes B) \rightarrow \mathbf{C}(A, B)$, which satisfies the following axioms (A1–A5, see also Fig. 4).

- (A1). **Tightening**, $u; \text{fbk}_S(f); v = \text{fbk}_S((\text{id} \otimes u); f; (\text{id} \otimes v))$.
- (A2). **Vanishing**, $\text{fbk}_I(f) = f$.
- (A3). **Joining**, $\text{fbk}_T(\text{fbk}_S(f)) = \text{fbk}_{S \otimes T}(f)$.
- (A4). **Strength**, $\text{fbk}_S(f) \otimes g = \text{fbk}_S(f \otimes g)$.
- (A5). **Sliding**, $\text{fbk}_T(f; (h \otimes \text{id})) = \text{fbk}_S((h \otimes \text{id}); f)$, for $h: S \rightarrow T$ any isomorphism.

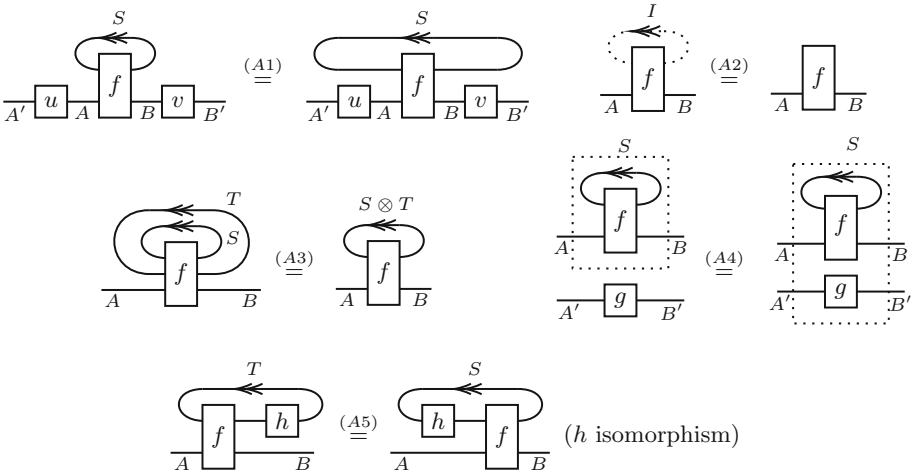


Fig. 4. Diagrammatic depiction of the axioms of feedback.

The natural notion of homomorphism between categories with feedback is that of a symmetric monoidal functor that moreover preserves the feedback structure. These are called *feedback functors*.

Definition 2.2. A feedback functor $F: \mathbf{C} \rightarrow \mathbf{D}$ between two *categories with feedback* $(\mathbf{C}, \text{fbk}^{\mathbf{C}})$ and $(\mathbf{D}, \text{fbk}^{\mathbf{D}})$ is a *strong symmetric monoidal functor* s.t.

$$F(\text{fbk}_S^{\mathbf{C}}(f)) = \text{fbk}_{F(S)}^{\mathbf{D}}(\mu; Ff; \mu^{-1}),$$

where $\mu_{A,B}: F(A) \otimes F(B) \rightarrow F(A \otimes B)$ is the isomorphism of the strong monoidal functor F . We write **Feedback** for the category of (small) *categories with feedback and feedback functors*. There is a forgetful functor $\mathcal{U}: \mathbf{Feedback} \rightarrow \mathbf{SymMon}$.

2.2 Traced Monoidal Categories

Categories with feedback are a weakening of the well-known *traced monoidal categories*. Between them, there is an intermediate notion called *right traced category* [39] that strengthens the sliding axiom from isomorphisms to arbitrary morphisms. This first extension would be already too strong for our purposes later in Sect. 2.4: we would be unable to define a *state space* up to isomorphism. However, the more conceptual difference of *traced monoidal categories* is the “yanking axiom” (in Fig. 5). Indeed, strengthening the *sliding axiom* and adding the *yanking axiom* yields the definition of *traced monoidal category*.

Traced monoidal categories are widely used in computer science. Since their conception [24] as an abstraction of the *trace* of a matrix in linear algebra, they were used in linear logic and geometry of interaction [1, 18, 19], programming language semantics [21], semantics of recursion [2] and fixed point operators [5, 22].

Traces are thus undeniably important, but it is questionable whether we really want to always impose *all* of their axioms. Specifically, we will be concerned with the *yanking axiom* that states that $\text{tr}(\sigma) = \text{id}$. The *yanking axiom* is incontestably elegant from the geometrical point of view: strings are “pulled”, and feedback (depicted as a loop with an arrowtip) disappears (Fig. 5).

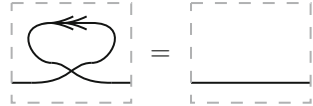


Fig. 5. The yanking axiom.

However, if feedback can disappear without leaving any imprint, that must mean that it is *instantaneous*: its output necessarily mirrors its input.² Importantly for our purposes, this implies that a feedback satisfying the yanking equation is “memoryless”, or “stateless”.

Consider again the NOR latch from Fig. 1. We have seen how to model NOR gates in **Span(Graph)** in Fig. 2, and the algebra of

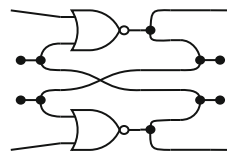


Fig. 6. Diagram for the NOR latch, modeled with a trace in **Span(Graph)**.

² In other words, traces are used to talk about processes in *equilibrium*, processes that have reached a *fixed point*. A theorem by Hasegawa [22] and Hyland [5] corroborates this interpretation: a trace in a cartesian category corresponds to a *fixpoint operator*.

Span(Graph) does include a trace (see Fig. 6, later detailed in Sect. 3.2). However, imitating the real-world behavior of the NOR latch with *just* a trace is unsatisfactory: the trace of **Span(Graph)** is built out of stateless components, and tracing stateless components yields a stateless component.

In engineering and computer science, instantaneous feedback is actually a rare concept; a more common notion is that of *guarded feedback*. Consider *signal flow graphs* [34, 40]: their categorical interpretation in [8] models feedback not by the usual trace, but by a trace “guarded by a register”, that *delays the signal* and violates the yanking axiom (see Remark 7.8 in *op.cit.*).

The component that trace misses in such examples is a *delay*.

2.3 Delay and Feedback

The main difference between **categories with feedback** and **traced monoidal categories** is the failure of the **yanking axiom**. Consider the process that only “feeds back” the input to itself and then uses that “fed back” input to produce the output. This process, $\partial_A := \text{fbk}_A(\sigma_{A,A})$, is called *delay endomorphism*. The *yanking axiom* of traced monoidal categories states that the delay is equal to the identity, $\text{tr}_A(\sigma_{A,A}) = \text{id}$, which is not necessarily true for *categories with feedback*. In that sense, a non-trivial delay is what sets apart categories with feedback from traced monoidal categories (Fig. 7).

This interpretation of feedback as the combination of *trace* and *delay* can be made into a theorem when the category has enough structure. *Compact closed categories* are **traced monoidal categories** where every object A has a dual A^* and the trace is constructed from two pieces $\varepsilon: A \otimes A^* \rightarrow I$ and $\eta: I \rightarrow A^* \otimes A$. While not every **traced monoidal category** is compact closed, they all embed fully faithfully into a compact closed category.³ In a **compact closed category**, a feedback operator is necessarily a trace “guarded” by a *delay*.

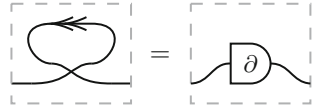


Fig. 7. Definition of *delay*.

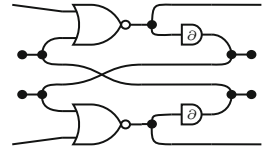


Fig. 8. NOR latch with feedback.

Proposition 2.3 (Feedback from delay [7]). *Let \mathbf{C} be a compact closed category with $\text{fbk}^{\mathbf{C}}$ a feedback operator that takes a morphism $S \otimes A \rightarrow S \otimes B$ to a morphism $A \rightarrow B$, satisfying the axioms of feedback (in Fig. 4) but possibly failing to satisfy the yanking axiom (Fig. 5) of traced monoidal categories. Then the feedback operator is necessarily of the form*

$$\text{fbk}_S^{\mathbf{C}}(f) := (\eta \otimes \text{id}); (\text{id} \otimes f); (\text{id} \otimes \partial_S \otimes \text{id}); (\varepsilon \otimes \text{id})$$

where $\partial_A: A \rightarrow A$ is a family of endomorphisms satisfying

³ This is the **Int** construction from [24].

- $\partial_A \otimes \partial_B = \partial_{A \otimes B}$ and $\partial_I = \text{id}$, and
- $\partial_A; h = h; \partial_B$ for each isomorphism $h: A \cong B$.

In fact, any family of morphisms ∂_A satisfying these properties determines uniquely a feedback operator that has ∂_A as its delay endomorphisms.

Consider again the NOR latch of Fig. 1. The algebra of **Span(Graph)** does include a feedback operator that is *not* a trace – the difference is an additional *stateful* delay component. As we shall see, this notion of feedback is canonical. We shall also see that the delay enables us to capture the real-world behavior of the NOR latch. The emergence of state from feedback is witnessed by the **St(•)** construction, which we recall below.

2.4 St(•), the Free Category with Feedback

Here we show how to obtain the free category with feedback over a symmetric monoidal category. The **St(•)** construction is a general way of endowing a system with state. It appears multiple times in the literature in slightly different forms: it constructs a stateful resource calculus in [7]; a variant is used for geometry of interaction in [23]; it coincides with the free **category with feedback** presented in [30]; and yet another, slightly different formulation was given in [26].

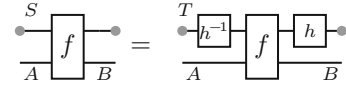


Fig. 9. Equivalence of stateful processes. We depict **stateful processes** by marking the space state.

Definition 2.4 (Category of stateful processes, [30]). Let (\mathbf{C}, \otimes, I) be a **symmetric monoidal category**. We write **St(C)** for the category with the objects of \mathbf{C} but where morphisms $A \rightarrow B$ are pairs $(S | f)$, consisting of a state space $S \in \mathbf{C}$ and a morphism $f: S \otimes A \rightarrow S \otimes B$. We consider morphisms up to isomorphism classes of their state space, and thus

$$(S | f) = (T | (h^{-1} \otimes \text{id}); f; (h \otimes \text{id})), \quad \text{for any isomorphism } h: S \cong T.$$

When depicting a **stateful process** (Fig. 9), we mark the state strings.

We define the **identity stateful process** on $A \in \mathbf{C}$ as $(I | \text{id}_{I \otimes A})$. **Sequential composition** of the two **stateful processes** $(S | f): A \rightarrow B$ and $(T | g): B \rightarrow C$ is defined by $(S | f); (T | g) = (S \otimes T | (\sigma \otimes \text{id}); (\text{id} \otimes f); (\sigma \otimes \text{id}); (\text{id} \otimes g))$. **Parallel composition** of the two **stateful processes** $(S | f): A \rightarrow B$ and $(S' | f'): A' \rightarrow B'$ is defined by $(S | f) \otimes (S' | f') = (S \otimes S' | (\text{id} \otimes \sigma \otimes \text{id}); (f \otimes f'); (\text{id} \otimes \sigma \otimes \text{id}))$ (Fig. 10).

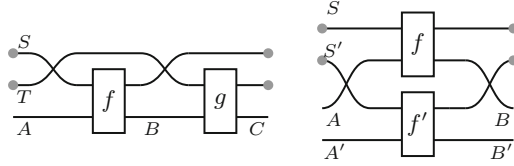


Fig. 10. Sequential and parallel composition of stateful processes.

This defines a symmetric monoidal category. Moreover, the operator

$$\text{store}_T(S \mid f) := (S \otimes T \mid f),$$

which “stores” some information into the state, makes it a category with feedback (Fig. 11).

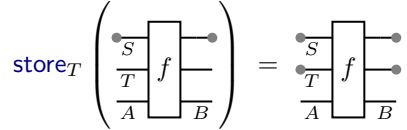


Fig. 11. The $\text{store}(\bullet)$ operation, diagrammatically.

Theorem 2.5 [30]. *The category $\text{St}(\mathbf{C})$, endowed with the $\text{store}(\bullet)$ operator, is the free category with feedback over a symmetric monoidal category \mathbf{C} .*

2.5 Examples

All traced monoidal categories are categories with feedback, since the axioms of feedback are a strict weakening of the axioms of trace. A more interesting source of examples is the $\text{St}(\bullet)$ construction we just defined.

Example 2.6. A Mealy deterministic transition system with boundaries A and B , and state space S was defined [35, §2.1] to be just a function $f: S \times A \rightarrow S \times B$.

It is not difficult to see that, up to isomorphism of the state space, they are morphisms of $\text{St}(\mathbf{Set})$. They compose following Definition 2.4, and form a category with feedback $\mathbf{Mealy} := \text{St}(\mathbf{Set})$.

The feedback of \mathbf{Mealy} transforms input/output pairs into states. Figure 12 is an example: a transition system with a single state becomes a transition system with two states and each transition $(s_i, i/s_o)$ yields a transition $(i/)$ from s_i to s_o .

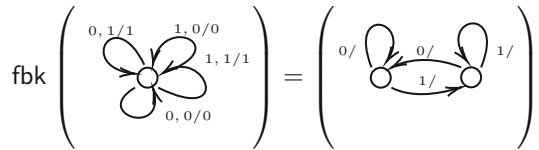


Fig. 12. Feedback of a Mealy transition system. Every transition has a label i/o indicating inputs (i) and outputs (o).

Similarly, when we consider \mathbf{Set} to be the monoidal structure of sets with the disjoint union, we recover Elgot automata [13], given by a transition function $S + A \rightarrow S + B$. These transition systems motivate the work in [26, 30].

Example 2.7. A linear dynamical system with inputs in \mathbb{R}^n , outputs in \mathbb{R}^m and state space \mathbb{R}^k is given by a matrix $\begin{pmatrix} A & B \\ C & D \end{pmatrix} \in \mathbf{Mat}_{\mathbb{R}}(k+m, k+n)$ [25]. Two linear dynamical systems $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$ and $\begin{pmatrix} A' & B' \\ C' & D' \end{pmatrix}$ are considered equivalent if there is an invertible matrix $H \in \mathbf{Mat}_{\mathbb{R}}(k, k)$ s.t. $A' = H^{-1}AH$, $B' = BH$, and $C' = H^{-1}C$.

Linear dynamical systems are morphisms of a category with feedback which coincides with $\mathbf{St}(\mathbf{Vect}_{\mathbb{R}}^{\oplus})$. The feedback operator is defined by

$$\text{fbk}_l \left(k, \begin{pmatrix} A_1 & A_2 & B_1 \\ A_3 & A_4 & B_2 \\ C_1 & C_2 & D \end{pmatrix} \right) = \left(k+l, \begin{pmatrix} A_1 & A_2 & B_1 \\ A_3 & A_4 & B_2 \\ C_1 & C_2 & D \end{pmatrix} \right),$$

where $\begin{pmatrix} A_1 & A_2 & B_1 \\ A_3 & A_4 & B_2 \\ C_1 & C_2 & D \end{pmatrix} \in \mathbf{Mat}_{\mathbb{R}}(k+l+m, k+l+n)$.

3 Span(Graph): An Algebra of Transition Systems

Span(Graph) [27] is an algebra of “open transition systems”. It has applications in *concurrency theory* and *verification* [17, 26, 27, 29, 31], and has been recently applied to biological systems [15, 16]. Just as ordinary Petri nets have an underlying (firing) semantics in terms of transition systems, **Span(Graph)** is used as a semantic universe for a variant of open Petri nets, see [9, 42].

An *open transition system* is a morphism of **Span(Graph)**: a transition graph endowed with two *boundaries* or *communication ports*. Each transition has an effect on each boundary, and this data is used for synchronization. This conceptual picture actually describes a subcategory, **Span(Graph)**_{*}, where boundaries are mere sets: the alphabets of synchronization signals. We shall recall the details of **Span(Graph)**_{*} and prove that it is universal, our main result:

Span(Graph)_{*} is the free *category with feedback* over **Span(Set)**.

3.1 The Algebra of Span(Graph)

Definition 3.1. A *span* [4, 10] from A to B , both objects of a category \mathbf{C} , is a pair of morphisms with a common domain, $A \leftarrow E \rightarrow B$. The object E is the “head” of the span, and the morphisms are the left and right “legs”, respectively.

When the category \mathbf{C} has pullbacks, we can sequentially compose two spans $A \leftarrow E \rightarrow B$ and $B \leftarrow F \rightarrow C$ obtaining $A \leftarrow E \times_B F \rightarrow C$. Here, $E \times_B F$ is the pullback of E and F along B : for instance, in **Set**, $E \times_B F$ is the subset of $E \times F$ given by pairs that have the same image in B .

Definition 3.2. Let \mathbf{C} be a category with pullbacks. **Span(C)** is the category that has the same objects as \mathbf{C} and isomorphism classes of spans between them as morphisms. That is, two spans are considered equal if there is an isomorphism between their heads that commutes with both legs. Dually, if \mathbf{C} is a category with pushouts, **Cospan(C)** is the category **Span(C^{op})**.

$\mathbf{Span}(\mathbf{C})$ is a **symmetric monoidal category** when \mathbf{C} has products. The parallel composition of $A \leftarrow E \rightarrow B$ and $A' \leftarrow E' \rightarrow B'$ is given by the componentwise product $A \times A' \leftarrow E \times E' \rightarrow B \times B'$. An example is again $\mathbf{Span}(\mathbf{Set})$.

Definition 3.3. *The category **Graph** has graphs $G = (s, t: E \rightrightarrows V)$ as objects, i.e. pairs of morphisms from edges to vertices returning the source and target of each edge. A morphism $G \rightarrow G'$ is given by functions $e: E \rightarrow E'$ and $v: V \rightarrow V'$ s.t. $e; s' = s; v$ and $e; t' = t; v$. Equivalently, it is the presheaf category on the diagram $(\bullet \rightrightarrows \bullet)$.*

We now focus on $\mathbf{Span}(\mathbf{Graph})_*$, those spans of graphs that have single node graphs ($A \rightrightarrows 1$) as the boundaries.

Definition 3.4. *An open transition system is a morphism of $\mathbf{Span}(\mathbf{Graph})_*$: a span of sets $A \leftarrow E \rightarrow B$ where the head is the set of transitions of a graph $E \rightrightarrows V$ (see Fig. 13). Two open transition systems are considered equal if there is an isomorphism between their graphs that commutes with the legs. Open transition systems whose graph $E \rightrightarrows 1$ has a single vertex are called stateless.*

Sequential composition (the *communicating-parallel operation* of [27]) of two open transition systems with spans $A \leftarrow E \rightarrow B$ and $B \leftarrow F \rightarrow C$ and graphs $E \rightrightarrows S$ and $F \rightrightarrows T$ yields the open transition system with span $A \leftarrow E \times_B F \rightarrow C$ and graph $E \times_B F \rightrightarrows S \times T$. This means that the only allowed transitions are those that synchronize E and F on the common boundary B .

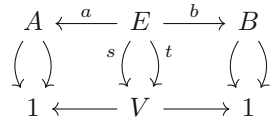


Fig. 13. A morphism of $\mathbf{Span}(\mathbf{Graph})_*$.

Parallel composition (the *non communicating-parallel operation* of [27]) of two open transition systems with spans $A \leftarrow E \rightarrow B$ and $A' \leftarrow E' \rightarrow B'$ and graphs $E \rightrightarrows V$ and $E' \rightrightarrows V'$ yields the open transition system with span $A \times A' \leftarrow E \times E' \rightarrow B \times B'$ and graph $E \times E' \rightrightarrows V \times V'$.

3.2 The Components of $\mathbf{Span}(\mathbf{Graph})$

Let us now detail some useful constants of the algebra of $\mathbf{Span}(\mathbf{Graph})_*$, which we will use to construct the NOR latch circuit from Fig. 8.

Example 3.5. The Frobenius algebra [10] ($\curvearrowright, \curvearrowleft, \bullet \rightarrow, \rightarrow \bullet$) is used for the “wiring”. The following spans are constructed out of diagonals $A \rightarrow A \times A$ and units $A \rightarrow 1$.

$$\begin{aligned} (\curvearrowright)_A &= \{A \leftarrow A \rightarrow A \times A\} & (\rightarrow \bullet)_A &= \{A \leftarrow A \rightarrow 1\} \\ (\curvearrowleft)_A &= \{A \times A \leftarrow A \rightarrow A\} & (\bullet \rightarrow)_A &= \{1 \leftarrow A \rightarrow A\} \end{aligned}$$

These induce a compact closed structure (and thus a trace), as follows:

$$(\bullet \curvearrowright)_A = \{1 \leftarrow A \rightarrow A \times A\} \quad (\curvearrowleft \bullet)_A = \{A \times A \leftarrow A \rightarrow 1\}.$$

In general, any function $f: A \rightarrow B$ can be lifted covariantly to a span $A \leftarrow A \rightarrow B$ and contravariantly to a span $A \leftarrow B \rightarrow B$. Any span $A \leftarrow E \rightarrow B$ can be lifted to $\text{Span}(\text{Graph})_*$ by making the head represent the graph $E \rightrightarrows 1$. We use this fact to obtain a stateless NOR gate from the function $\text{NOR}: \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$ (Fig. 2).

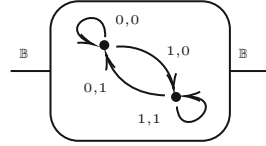


Fig. 14. Delay morphism over the set $\mathbb{B} := \{0, 1\}$.

We will need a single stateful component to model our circuit, the delay

$$(\dashv\!\!\!\dashv)A = \left\{ \begin{array}{c} A \times A \\ \swarrow \pi_2 \quad \downarrow \pi_1 \quad \searrow \pi_2 \\ A \quad A \quad A \end{array} \right\}.$$

This is *not* an arbitrary choice: it is the canonical delay obtained from the feedback structure⁴ in $\text{Span}(\text{Graph})_*$ (Fig. 14).

The NOR latch circuit of Fig. 8 is the composition of two NOR gates where the outputs of each gate have been copied and fed back as input to the other gate. The algebraic expression, in $\text{Span}(\text{Graph})_*$, of this circuit is obtained by decomposing it into its components, as in Fig. 15.

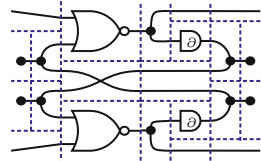


Fig. 15. Decomposing the circuit.

$$(\text{id} \otimes \bullet \curvearrowright \bullet \curvearrowright \otimes \text{id}); (\text{NOR} \otimes \sigma \otimes \text{NOR}); (\bullet \curvearrowright \otimes \text{id} \otimes \bullet \curvearrowright); (\text{id} \otimes \dashv\!\!\!\dashv \otimes \text{id} \otimes \dashv\!\!\!\dashv \otimes \text{id}); (\text{id} \otimes \bullet \curvearrowright \bullet \curvearrowright \otimes \text{id})$$

The graph obtained from this expression, together with its transitions, is shown in Fig. 16. This time, our model is indeed stateful. It has four states: two states representing a correctly stored signal, $\bar{A} = (1, 0)$ and $A = (0, 1)$; and two states representing transitory configurations $T_1 = (0, 0)$ and $T_2 = (1, 1)$.

The *left boundary* can receive a *set* signal, $\text{Set} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$; a *reset* signal, $\text{Reset} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$; none of the two, $\text{Idle} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$; or both of them at the same time, $\text{Unspec} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, which is known to cause unspecified behavior in a NOR latch.

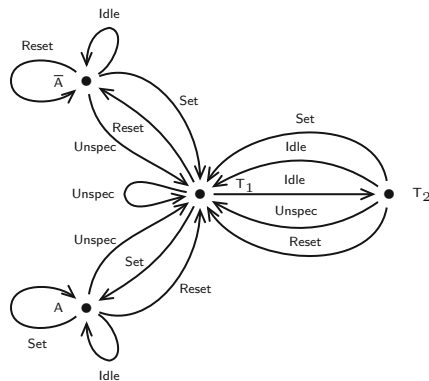


Fig. 16. Span of graphs representing the NOR latch

⁴ As in Sect. 2, $\partial_A = \text{fbk}(\sigma_{A,A})$.

The signal on the *right boundary*, on the other hand, is always equal to the state the transition goes to and does not provide any additional information: we omit it from Fig. 16.

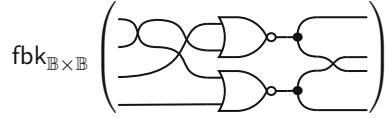


Fig. 17. Applying $\text{fbk}(\bullet)$ over the circuit gives the NOR latch.

Activating the signal *Set* makes the latch reach the state *A* in (at most) two transition steps. Activating *Reset* does the same for \bar{A} . After any of these two cases, deactivating all signals, *Idle*, keeps the last state.

Moreover, the (real-world) NOR latch has some unspecified behavior that gets also reflected in the graph: activating both *Set* and *Reset* at the same time, what we call *Unspec*, causes the circuit to enter an unstable state where it bounces between the states T_1 and T_2 after an *Idle* signal. Our modeling has reflected this “unspecified behavior” as expected.

Feedback and Trace. In terms of feedback, the circuit of Fig. 16 is equivalently obtained as the result of taking feedback over the stateless morphism in Fig. 17.

But $\text{Span}(\text{Graph})_*$ is also canonically traced: it is actually compact closed. What changes in the modeling if we would have used the trace instead? As we argued for Fig. 6, we obtain a stateless transition system. The valid transitions are

$$\{(\text{Unspec}, T_1), (\text{Idle}, A), (\text{Idle}, \bar{A}), (\text{Set}, A), (\text{Reset}, \bar{A})\}.$$

They encode important information: they are the *equilibrium* states of the circuit. However, unlike the previous graph, this one would not get us the correct allowed transitions: under this modeling, our circuit could freely bounce between (Idle, A) and (Idle, \bar{A}) , which is not the expected behavior of a NOR latch.

The fundamental piece making our modeling succeed the previous time was feedback with *delay*. Next we show that this feedback is canonical.

3.3 Span(Graph) as a Category with Feedback

This section presents our main theorem. We introduce a mapping that associates to each stateful span of sets a corresponding span of graphs. This mapping is well-defined and lifts to a functor $\text{St}(\text{Span}(\text{Set})) \rightarrow \text{Span}(\text{Graph})$. Finally, we prove that it is an isomorphism $\text{St}(\text{Span}(\text{Set})) \cong \text{Span}(\text{Graph})_*$.

Proposition 3.6. *The composition of two stateful spans in $\text{St}(\text{Span}(\text{Set}))$,*

$$S \times A \xleftarrow{\sigma, f} X \xrightarrow{\sigma', g} S \times B, \quad T \times B \xleftarrow{\tau, h} Y \xrightarrow{\tau', k} T \times C$$

is the span $T \times S \times A \xleftarrow{\tau, \sigma, f} X \times_B Y \xrightarrow{\tau', \sigma', k} T \times S \times C$, where $X \times_B Y$ is the pullback along g and h .

Lemma 3.7. *The following assignment of **stateful processes** over $\mathbf{Span}(\mathbf{Set})$ to morphisms of $\mathbf{Span}(\mathbf{Graph})$ is well defined.*

$$K \left(S \left| \begin{array}{ccc} & E & \\ (s,a) \swarrow & & \searrow (t,b) \\ S \times A & & S \times B \end{array} \right. \right) := \left(\begin{array}{ccc} A & \xleftarrow{a} E & \xrightarrow{b} B \\ \left(\downarrow \right) & s \left(\downarrow \right) t & \left(\downarrow \right) \\ 1 & \longleftarrow S & \longrightarrow 1 \end{array} \right)$$

Theorem 3.8. *There exists an isomorphism of categories $\mathbf{St}(\mathbf{Span}(\mathbf{Set})) \cong \mathbf{Span}(\mathbf{Graph})_*$. That is, the free category with feedback over $\mathbf{Span}(\mathbf{Set})$ is isomorphic to the full subcategory of $\mathbf{Span}(\mathbf{Graph})$ given by single-vertex graphs.*

Proof. We prove that there is a fully faithful functor $K: \mathbf{St}(\mathbf{Span}(\mathbf{Set})) \rightarrow \mathbf{Span}(\mathbf{Graph})$ defined on objects as $K(A) = (A \rightrightarrows 1)$ and defined on morphisms as in Lemma 3.7.

We now show that K is functorial, preserving composition and identities. We can directly see that the identity morphism in $\mathbf{St}(\mathbf{Span}(\mathbf{Set}))$, as a span $1 \times A \leftarrow A \rightarrow 1 \times A$, is sent to the identity span of the graph $A \rightrightarrows 1$.

Let us now show that composition is also preserved. Let us consider two stateful spans; the first given by $s, a: E \rightarrow S \times A$ and $s', b: E \rightarrow S \times B$; the second given by $t, b': F \rightarrow T \times B$ and $t', c: F \rightarrow T \times C$. Their composition is given by a span whose legs are $s, t, a: E \times_B F \rightarrow S \times T \times A$ and $s', t', c: E \times_B F \rightarrow S \times T \times C$, and $E \times_B F$ is the pullback along b and b' (see Proposition 3.6).

We have composed two stateful spans; and we want to show that the corresponding graph of the composite is the pullback of their graphs. Computing a pullback of graphs can be done separately on edges and vertices, as graphs form a presheaf category (see Fig. 18). Note how the resulting graph is precisely the graph corresponding, under the assignment K , to the previous stateful span.

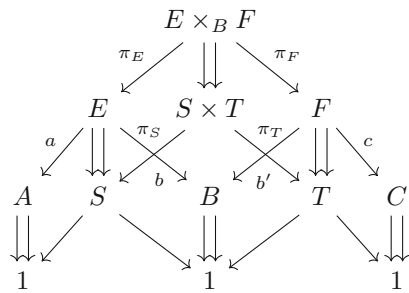


Fig. 18. Pullback of graphs.

The final step is to show that the original assignment is fully-faithful. We can see that it is full: every span of single-vertex graphs given by $A \leftarrow E \rightarrow B$ and $E \rightrightarrows S$ does arise from some span, namely $S \times A \leftarrow E \rightarrow S \times B$. Let us check it is also faithful. Suppose that two morphisms in $\mathbf{St}(\mathbf{Span}(\mathbf{Set}))$, $S \times A \leftarrow E \rightarrow S \times B$ and $S' \times A \leftarrow E' \rightarrow S' \times B$, are sent to equivalent spans of graphs, i.e. there exist $h: E \cong E'$ and $k: S' \cong S$ making the diagrams in Fig. 19 commute.

In this case, we know that $S \times A \leftarrow E \rightarrow S \times B$ is equivalent to $S' \times A \leftarrow E \rightarrow S' \times B$ because of the equivalence relation on [stateful processes](#). Finally, $S' \times A \leftarrow E \rightarrow S' \times B$ is equivalent as a span to $S' \times A \leftarrow E' \rightarrow S' \times B$.

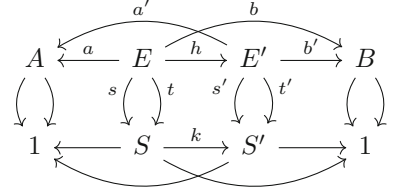


Fig. 19. Equivalent spans of graphs.

We have shown that there exists a fully-faithful functor from the free category with feedback over [Span\(Set\)](#) to the category [Span\(Graph\)](#) of spans of graphs. The functor restricts to an equivalence between [St\(Span\(Set\)\)](#) and the full subcategory of [Span\(Graph\)](#) on single-vertex graphs. It is moreover bijective on objects, giving an isomorphism of categories. \square

Example 3.9. The characterization $\text{Span(Graph)}_* \cong \text{St(Span(Set))}$ that we prove in Theorem 3.8 lifts the inclusion $\text{Set} \rightarrow \text{Span(Set)}$ to a feedback preserving functor $\text{Mealy} \rightarrow \text{Span(Graph)}_*$. This inclusion embeds a deterministic transition system into the graph that determines it.

3.4 Cospan(Graph) as a Category with Feedback

Theorem 3.8 can be generalized to any category \mathbf{C} with finite limits. By taking $\text{Graph}(\mathbf{C})$ to be the presheaf category of the diagram $(\bullet \rightrightarrows \bullet)$ in \mathbf{C} and $\text{Span(Graph}(\mathbf{C}))_*$ the full subcategory on objects of the form $A \rightrightarrows 1$, we have:

Theorem 3.10. *There exists an isomorphism of categories $\text{St(Span}(\mathbf{C})) \cong \text{Span(Graph}(\mathbf{C}))_*$. That is, the free category with feedback over [Span\(C\)](#) is equivalent to the full subcategory on [Span\(Graph\(C\)\)](#) given by single-vertex graphs.*

Cospan(Graph)_* can be also characterized as a free category with feedback. We know that $\text{Cospan(Set)} \cong \text{Span(Set}^{op})$, we note that $\text{Graph(Set}^{op}) \cong \text{Graph}^{op}(\text{Set})$ (which has the effect of flipping edges and vertices), and we can use Theorem 3.10 because Set has all finite colimits. The explicit assignment is similar to the one shown in Lemma 3.7.

$$K \left(S \left| \begin{array}{ccc} & S & \\ [t|a] \nearrow & & \nwarrow [s|b] \\ E + A & & E + B \end{array} \right. \right) := \left(\begin{array}{ccc} A & \xrightarrow{a} & S \xleftarrow{b} B \\ \left(\uparrow \right) & & t \left(\uparrow \right) s \left(\uparrow \right) \\ 0 & \longrightarrow & E \longleftarrow 0 \end{array} \right)$$

Corollary 3.11. *There is an isomorphism $\text{St(Cospan(Set))} \cong \text{Cospan(Graph)}_*$.*

Cospan(Graph) is also compact closed and, in particular, traced. As in the case of Span(Graph) , the feedback structure given by the universal property is different from the trace. In the case of Cospan(Graph) , tracing has the effect of identifying the output and input vertices of the graph; while feedback adds an additional edge from the output to the input vertices.

3.5 Syntactical Presentation of $\mathbf{Cospan}(\mathbf{FinGraph})$

The observation in Proposition 2.3 has an important consequence in the case of finite sets. We write $\mathbf{FinGraph}$ for $\mathbf{Graph}(\mathbf{FinSet})$. $\mathbf{Cospan}(\mathbf{FinSet})$ is the generic special commutative Frobenius algebra [32], meaning that any morphism written out of the operations of a special commutative Frobenius algebra and the structure of a symmetric monoidal category is precisely a cospan of finite sets. But we also know that $\mathbf{Cospan}(\mathbf{FinSet})$, with an added generator to its PROP structure [7] is $\mathbf{St}(\mathbf{Cospan}(\mathbf{FinSet}))$, or, equivalently, $\mathbf{Cospan}(\mathbf{FinGraph})$. This means that any morphism written out of the operations of a special commutative Frobenius algebra plus a freely added generator of type $(-\mathbb{D}-): 1 \rightarrow 1$ is a morphism in $\mathbf{Cospan}(\mathbf{FinGraph})_*$. This way, we recover one of the main results of [37] as a direct corollary of our characterization.

Proposition 3.12 (Proposition 3.2 of [37]). $\mathbf{Cospan}(\mathbf{FinGraph})_*$ is the generic special commutative Frobenius monoid with an added generator.

Proof. It is known that the category $\mathbf{Cospan}(\mathbf{FinSet})$ is the generic special commutative Frobenius algebra [32]. Adding a free generator $(-\mathbb{D}-): 1 \rightarrow 1$ to its PROP structure corresponds to adding a family $(-\mathbb{D}-)_n: n \rightarrow n$ with the conditions on Proposition 2.3. Now, Proposition 2.3 implies that adding such a generator to $\mathbf{Cospan}(\mathbf{FinSet})$ results in $\mathbf{St}(\mathbf{Cospan}(\mathbf{FinSet}))$. Finally, we use Theorem 3.8 to conclude that $\mathbf{St}(\mathbf{Cospan}(\mathbf{FinSet})) \cong \mathbf{Cospan}(\mathbf{FinGraph})_*$. \square

4 Conclusions and Further Work

We characterized $\mathbf{Span}(\mathbf{Graph})_*$, an algebra of open transition systems, as the free *category with feedback* over the category of spans of functions. To do so, we use the $\mathbf{St}(\bullet)$ construction, characterized as the free *category with feedback* in [30]. It is also well-known as a technique of adding state to processes.

We have seen how the $\mathbf{St}(\bullet)$ construction creates categories of *transition systems* out of symmetric monoidal categories. We could also consider a generalization of this construction where, instead of quotienting by isomorphisms, we can quotient by arbitrary classes of morphisms selected by some strong monoidal functor. Our observation is that this generalized state construction can be rewritten compactly as a particular kind of colimit called a *coend* (see [33] for a definition). In fact, let $F: \mathbf{D} \rightarrow \mathbf{C}$ be a strong monoidal functor, we can express the set of stateful morphisms quotiented by *sliding* in \mathbf{D} as

$$\mathbf{St}_{\mathbf{D}}(\mathbf{C}) := \int^{D \in \mathbf{D}} \mathbf{hom}(FD \otimes X, FD \otimes Y).$$

For instance, the original $\mathbf{St}(\bullet)$ construction is recovered from the inclusion functor of the subgroupoid of isomorphisms (also known as the “*core*” of the category). The identity functor can be used to quotient processes by dinaturality. The forgetful $\mathbf{PointedSet} \rightarrow \mathbf{Set}$ can be used to construct automata with initial states. We plan to investigate the relationship between such generalized *categories with feedback* to approaches based on guarded recursion [20] and coalgebras [11, 36].

References




1. Abramsky, S.: What are the fundamental structures of concurrency? We still don't know! CoRR abs/1401.4973 (2014). <http://arxiv.org/abs/1401.4973>
2. Adámek, J., Milius, S., Velebil, J.: Elgot algebras. *Log. Methods Comput. Sci.* **2**(5) (2006). [https://doi.org/10.2168/LMCS-2\(5:4\)2006](https://doi.org/10.2168/LMCS-2(5:4)2006)
3. Baez, J.C., Courser, K.: Structured cospans. CoRR abs/1911.04630 (2019)
4. Bénabou, J.: Introduction to bicategories. In: Reports of the Midwest Category Seminar. LNM, vol. 47, pp. 1–77. Springer, Heidelberg (1967). <https://doi.org/10.1007/BFb0074299>
5. Benton, N., Hyland, M.: Traced premonoidal categories. *RAIRO Theor. Inform. Appl.* **37**(4), 273–299 (2003). <https://doi.org/10.1051/ita:2003020>
6. Bloom, S.L., Ésik, Z.: Iteration Theories - The Equational Logic of Iterative Processes. EATCS Monographs on Theoretical Computer Science. Springer, Heidelberg (1993). <https://doi.org/10.1007/978-3-642-78034-9>
7. Bonchi, F., Holland, J., Piedeleu, R., Sobociński, P., Zanasi, F.: Diagrammatic algebra: from linear to concurrent systems. *Proc. ACM Program. Lang.* **3**(POPL), 25:1–25:28 (2019). <https://doi.org/10.1145/3290338>
8. Bonchi, F., Sobociński, P., Zanasi, F.: The calculus of signal flow diagrams I: linear relations on streams. *Inf. Comput.* **252**, 2–29 (2017). <https://doi.org/10.1016/j.ic.2016.03.002>
9. Bruni, R., Melgratti, H., Montanari, U.: A connector algebra for P/T nets interactions. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 312–326. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23217-6_21
10. Carboni, A., Walters, R.F.C.: Cartesian bicategories I. *J. Pure Appl. Algebra* **49**(1–2), 11–32 (1987)
11. Clouston, R., Bizjak, A., Grathwohl, H.B., Birkedal, L.: Programming and reasoning with guarded recursion for coinductive types. In: Pitts, A. (ed.) FoSSaCS 2015. LNCS, vol. 9034, pp. 407–421. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46678-0_26
12. Eccles, W.H., Jordan, F.W.: Improvements in ionic relays. British patent number: GB 148582 (1918)
13. Elgot, C.C.: Monadic computation and iterative algebraic theories. In: *Studies in Logic and the Foundations of Mathematics*, vol. 80, pp. 175–230. Elsevier (1975)
14. Fong, B.: Decorated cospans. *Theory Appl. Categories* **30**(33), 1096–1120 (2015)
15. Gianola, A., Kasangian, S., Manicardi, D., Sabadini, N., Schiavio, F., Tini, S.: CospanSpan(Graph): a compositional description of the heart system. *Fundam. Informaticae* **171**(1–4), 221–237 (2020)
16. Gianola, A., Kasangian, S., Manicardi, D., Sabadini, N., Tini, S.: Compositional modeling of biological systems in CospanSpan(Graph). In: *Proceedings of ICTCS 2020*. CEUR-WS (2020, to appear)
17. Gianola, A., Kasangian, S., Sabadini, N.: Cospan/Span(Graph): an algebra for open, reconfigurable automata networks. In: Bonchi, F., König, B. (eds.) 7th Conference on Algebra and Coalgebra in Computer Science, CALCO 2017, Ljubljana, Slovenia, 12–16 June 2017. *LIPICs*, vol. 72, pp. 2:1–2:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). <https://doi.org/10.4230/LIPICs.CALCO.2017.2>
18. Girard, J.: Linear logic. *Theor. Comput. Sci.* **50**, 1–102 (1987). [https://doi.org/10.1016/0304-3975\(87\)90045-4](https://doi.org/10.1016/0304-3975(87)90045-4)

19. Girard, J.Y.: Towards a geometry of interaction. *Contemp. Math.* **92**(69–108), 6 (1989)
20. Goncharov, S., Schröder, L.: Guarded traced categories. In: Baier, C., Dal Lago, U. (eds.) *FoSSaCS 2018*. LNCS, vol. 10803, pp. 313–330. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89366-2_17
21. Hasegawa, M.: Recursion from cyclic sharing: traced monoidal categories and models of cyclic lambda calculi. In: de Groote, P., Roger Hindley, J. (eds.) *TLCA 1997*. LNCS, vol. 1210, pp. 196–213. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-62688-3_37
22. Hasegawa, M.: The uniformity principle on traced monoidal categories. In: Blute, R., Selinger, P. (eds.) *Category Theory and Computer Science, CTCS 2002*, Ottawa, Canada, 15–17 August 2002. *Electronic Notes in Theoretical Computer Science*, vol. 69, pp. 137–155. Elsevier (2002). [https://doi.org/10.1016/S1571-0661\(04\)80563-2](https://doi.org/10.1016/S1571-0661(04)80563-2)
23. Hoshino, N., Muroya, K., Hasuo, I.: Memoryful geometry of interaction: from coalgebraic components to algebraic effects. In: Henzinger, T.A., Miller, D. (eds.) *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS 2014*, Vienna, Austria, 14–18 July 2014, pp. 52:1–52:10. ACM (2014). <https://doi.org/10.1145/2603088.2603124>
24. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. *Math. Proc. Cambridge Philos. Soc.* **119**, 447–468 (1996). <https://doi.org/10.1017/S0305004100074338>
25. Kalman, R.E., Falb, P.L., Arbib, M.A.: *Topics in Mathematical System Theory*, vol. 1. McGraw-Hill, New York (1969)
26. Katis, P., Sabadini, N., Walters, R.F.C.: Bicategories of processes. *J. Pure Appl. Algebra* **115**(2), 141–178 (1997)
27. Katis, P., Sabadini, N., Walters, R.F.C.: Span(Graph): a categorical algebra of transition systems. In: Johnson, M. (ed.) *AMAST 1997*. LNCS, vol. 1349, pp. 307–321. Springer, Heidelberg (1997). <https://doi.org/10.1007/BFb0000479>
28. Katis, P., Sabadini, N., Walters, R.F.C.: On the algebra of feedback and systems with boundary. In: *Rendiconti del Seminario Matematico di Palermo* (1999)
29. Katis, P., Sabadini, N., Walters, R.F.C.: A formalization of the IWIM model. In: Porto, A., Roman, G.-C. (eds.) *COORDINATION 2000*. LNCS, vol. 1906, pp. 267–283. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45263-X_17
30. Katis, P., Sabadini, N., Walters, R.F.C.: Feedback, trace and fixed-point semantics. *RAIRO-Theor. Inform. Appl.* **36**(2), 181–194 (2002). <https://doi.org/10.1051/ita:2002009>
31. Katis, P., Sabadini, N., Walters, R.F.C.: A process algebra for the Span(Graph) model of concurrency. arXiv preprint [arXiv:0904.3964](https://arxiv.org/abs/0904.3964) (2009)
32. Lack, S.: Composing PROPs. *Theory Appl. Categories* **13**(9), 147–163 (2004)
33. Mac Lane, S.: *Categories for the Working Mathematician*. Graduate Texts in Mathematics, Springer, New York (1978). <https://doi.org/10.1007/978-1-4757-4721-8>
34. Mason, S.J.: Feedback theory - some properties of signal flow graphs. *Proc. Inst. Radio Eng.* **41**(9), 1144–1156 (1953). <https://doi.org/10.1109/JRPROC.1953.274449>
35. Mealy, G.H.: A method for synthesizing sequential circuits. *Bell Syst. Tech. J.* **34**(5), 1045–1079 (1955)
36. Milius, S., Litak, T.: Guard your daggers and traces: properties of guarded (co-) recursion. *Fund. Inform.* **150**(3–4), 407–449 (2017)

37. Rosebrugh, R., Sabadini, N., Walters, R.F.C.: Generic commutative separable algebras and cospans of graphs. *Theory Appl. Categories* **15**(6), 164–177 (2005)
38. Sabadini, N., Schiavio, F., Walters, R.F.C.: On the geometry and algebra of networks with state. *Theor. Comput. Sci.* **664**, 144–163 (2017)
39. Selinger, P.: A survey of graphical languages for monoidal categories. In: Coecke B. (ed.) *New Structures for Physics*, vol. 813, pp. 289–355. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12821-9_4
40. Shannon, C.E.: *The Theory and Design of Linear Differential Equation Machines*. Bell Telephone Laboratories (1942)
41. Sobociński, P.: A non-interleaving process calculus for multi-party synchronisation. In: *2nd Interaction and Concurrency Experience: Structured Interactions*, (ICE 2009). *EPTCS*, vol. 12 (2009). <https://doi.org/10.4204/eptcs.12.6>. <http://users.ecs.soton.ac.uk/ps/papers/ice09.pdf>
42. Sobociński, P.: Representations of Petri net interactions. In: Gastin, P., Laroussinie, F. (eds.) *CONCUR 2010*. *LNCS*, vol. 6269, pp. 554–568. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_38



Corinne, a Tool for Choreography Automata

Simone Orlando¹, Vairo Di Pasquale¹, Franco Barbanera²,
Ivan Lanese³, and Emilio Tuosto⁴

¹ University of Bologna, Bologna, Italy

² Department of Mathematics and Computer Science, University of Catania,
Catania, Italy

`barba@dmi.unict.it`

³ Focus Team, University of Bologna/INRIA, Bologna, Italy

⁴ Gran Sasso Science Institute, L'Aquila, Italy

`emilio.tuosto@gssi.it`

Abstract. Choreography automata are a model of choreographies envisaging high-level views of the behaviour of communicating systems as finite-state automata. The behaviour of each participant of a choreography can be obtained via a projection operation from a choreography automaton. The system of participants obtained by projection is well-behaved if the choreography automaton satisfies some well-formedness conditions. We present Corinne, a tool allowing one to render, compute projections of and compose choreography automata, as well as to check well-formedness conditions.

1 Introduction

Programming and understanding distributed systems is notoriously difficult due to the need to reason on multiple flows of execution and many possible behaviours; yet distributed systems are fundamental nowadays. Indeed most of our systems, from social networks to apps, from games to scientific software, are distributed. A main challenge when programming distributed systems, in particular multiparty ones, is how to define communication protocols avoiding subtle bugs such as deadlocks.

In order to reason on the correctness and properties of multiparty communication protocols, dedicated models such as conversation protocols [24], choreographies [11, 28, 31], global graphs [35], and multiparty session types [12, 26, 27]

Research partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233, by the MIUR project PRIN 2017FTXR7S “IT-MaTTerS” (Methods and Tools for Trustworthy Smart Systems), and by the Progetto di Ateneo Pia.Ce.Ri - UNICT. The third and fourth authors have also been partially supported by INdAM as members of GNCS (Gruppo Nazionale per il Calcolo Scientifico). The authors thank the reviewers for their interesting comments and suggestions, which helped us to improve the paper. The third author wishes to thank also Mariangiola Dezani-Ciancaglini for her support.

© Springer Nature Switzerland AG 2021

G. Salaün and A. Wijs (Eds.): FACS 2021, LNCS 13077, pp. 82–92, 2021.

https://doi.org/10.1007/978-3-030-90636-8_5

have been proposed. Their common trait is to provide global descriptions of the behaviour of a distributed system, and to allow one to ensure desirable properties such as deadlock freedom by checking some structural conditions on the model. Also, they provide an operation, called projection, to extract from the global specification a description of the (local) behaviour that each participant has to follow in order to implement the desired global behaviour.

In this paper we focus on choreography automata (c-automata) [4], which are an automata model belonging to the family described above. Essentially, c-automata are finite-state automata whose transitions are labelled by interactions representing point-to-point communications between a sender and a receiver. Despite its simplicity, manually performing the constructions and analysis (such as the ones in [4]) on c-automata is tedious, error prone even for simple cases, and its complexity increases with the size of c-automata to the point that it becomes practically impossible even on moderately large instances.

Thus, we decided to automate the main constructions and analysis on c-automata in a prototype tool called **Corinne**. This tool allows us to experiment with c-automata. We illustrate the usefulness of **Corinne** by applying it to the examples in [3, 4]. This exercise allowed us to spot a couple of (minor) errors in [4]. We will come back to this when describing the tool. It is worth noticing that the definitions of choreography automaton and of its projection are independent of the chosen communication model (synchronous or asynchronous). Indeed, this choice affects only the definition of well-formedness conditions.

After reviewing the main constructions and operations of c-automata (Sect. 2), we introduce **Corinne** (Sect. 3). We will conclude the paper (Sect. 4) with some final remarks.

Corinne is available at [14] (under the open-source MIT license), together with all the examples discussed in this paper.

2 Choreography Automata

This section surveys choreography automata (c-automata) borrowing definitions and concepts from [4, 5]; for full details about this formalism we refer the reader to [4, 5]. C-automata (ranged over by CA, CB, etc.) are Finite-State Automata (FSAs) whose transitions are labelled by interactions of the form $A \rightarrow B : m$; such interaction represents a communication between participants **A** and **B** where the former sends a message (of type) **m** to the latter, which is supposed to receive **m**. We let λ range over the set \mathcal{L}_{int} of interactions.

Definition 2.1 (Choreography automata). *A choreography automaton (c-automaton) is an FSA on the alphabet \mathcal{L}_{int} , namely a tuple $\langle Q, q_0, \mathcal{L}_{int}, \rightarrow \rangle$ where Q is a finite set of states, q_0 the initial state, and $\rightarrow \subseteq Q \times \mathcal{L}_{int} \times Q$ the transition relation. We write $q \xrightarrow{\lambda} q'$ when $(q, \lambda, q') \in \rightarrow$.*

Given a c-automaton, the projection operation builds the corresponding communicating system consisting of the set of projections of the c-automaton on each participant. Each projection is an FSA as well, on the alphabet \mathcal{L}_{act} of actions,

which have the form $AB!m, AB?m$. The former denotes the action of sending message m from A to B , the latter the corresponding receiving action. Such FSAs are called Communicating Finite State Machines (CFSMs) [10]. Hereafter, \mathcal{P}_{CA} denotes the set of participants of a c-automaton CA ; note that \mathcal{P}_{CA} is necessarily finite.

Definition 2.2 (Automata projection). *The projection on A of a transition $t = q \xrightarrow{\lambda} q'$ of a c-automaton, written $t \downarrow_A$, is defined by:*

$$t \downarrow_A = \begin{cases} q \xrightarrow{AB!m} q' & \text{if } \lambda = A \rightarrow B : m \\ q \xrightarrow{BA?m} q' & \text{if } \lambda = B \rightarrow A : m \\ q \xrightarrow{\epsilon} q' & \text{otherwise} \end{cases}$$

The projection of a c-automaton $CA = \langle Q, q_0, \mathcal{L}_{int}, \rightarrow \rangle$ on a participant $A \in \mathcal{P}_{CA}$, denoted $CA \downarrow_A$, is obtained by determinising¹ up-to-language equivalence the intermediate automaton

$$A_A = \langle Q, q_0, \mathcal{L}_{act} \cup \{\epsilon\}, \{(q \xrightarrow{\lambda} q') \downarrow_A \mid q \xrightarrow{\lambda} q'\} \rangle$$

The projection of CA , written $CA \downarrow$, is the communicating system $(CA \downarrow_A)_{A \in \mathcal{P}_{CA}}$.

The projection of c-automata is essentially obtained by transferring projections of global specifications present in several choreography-based approaches such as, e.g., [12, 13, 26, 27, 35]. A composition operation on c-automata has been recently proposed by the last three authors in [5]. The idea is to *lift* at the choreographic level a version of the composition of systems of CFSMs described in [2] and applied in a multiparty session type setting in [3]. This technique enables to overcome the fact that in choreographic approaches systems are usually intended to be *closed*. Actually, it is instead possible to look at any system as an *open* one (so enabling modular development) by looking at any of its participants as a possible interface. Hence, the composition of systems is essentially obtained by taking two systems, selecting two of their participants (one per system) provided that they meet some *compatibility* conditions, and removing them while redirecting communications to them towards the other system. More precisely, if a message is sent by some participant A to the chosen interface of the system it belongs to, compatibility conditions require the interface of the other system to send an identical message to some participant B . In the composed system A sends the message directly to B . This way of composing systems can be obtained by applying one after the other two operations:

1. the product of c-automata, building a c-automaton corresponding to the concurrent execution of the two original c-automata; and
2. a *blending* operation that, given two participants (the chosen interfaces) of a same c-automaton, removes them and adjusts the c-automaton as described above.

¹ In [4] also minimisation is performed, but this is not needed for the correctness of the constructions, and it is not currently performed by [Corinne](#).

The formalisation of these operations can be found in [5], while an example will be discussed in the next section (Fig. 3).

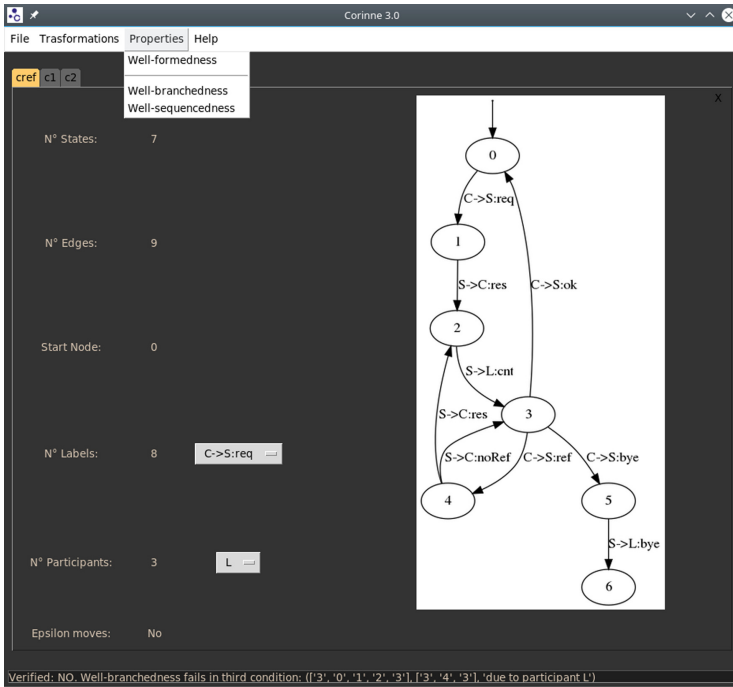


Fig. 1. Corinne screenshot

3 Corinne

The operations of projection and composition of c -automata described in Sect. 2 are implemented in Corinne [14]. The tool is written in python3 and works on c -automata represented as particular directed graphs in the DOT format [23]. Rendering of DOT files is performed using the `graphviz` library [25]. Other formats can be used as input of Corinne; more precisely the tool also parses regular expressions used as the syntax of global graphs [35] in ChorGram [15, 17], or the DOT representation [23] of global graphs produced by Domitilla [22]. We remark that only global graphs with no parallel composition correspond to c -automata and can thus be imported. All parsers are defined using ANTLR4 [32].

Users interact with **Corinne** through a graphical interface based on the **tkinter** package [34]. The GUI of **Corinne** displays FSAs that are either c-automata or CFSMs obtained via projection. As shown in the screenshot in Fig. 1, each FSA appears in a separate tab. The tab also reports basic information on the FSA (e.g., number of states and of edges) as well as a graphical rendering of the FSA itself.

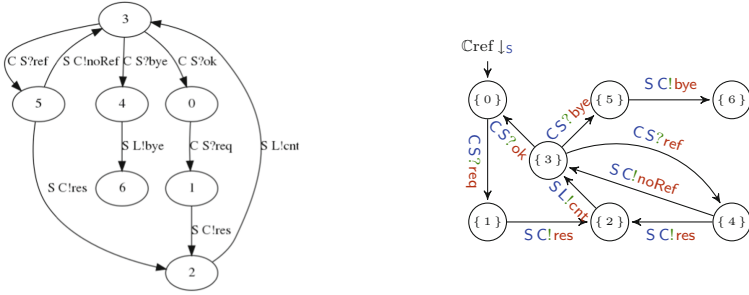


Fig. 2. Projections of $\mathbb{C}ref$ on \mathbb{S} from **Corinne** (left) and [4] (right)

Besides utility menus **File** and **Help**, **Corinne** has two menus to work on c-automata. Menu **Transformations** allows one to compute projections² on a given participant, the product of two c-automata as well as the blending (synchronisation) operation via interfaces following the approach described in [5].

Menu **Properties** instead allows one to check the well-formedness conditions discussed in [4] ensuring that the language of the c-automaton coincides with the one of the (synchronous) system obtained via projection, and that the latter is live, lock-free, and deadlock-free (we refer to [4] for the definition of these properties in the context of c-automata as well as for formal statements and proofs of the results hinted at above).

The screenshot in Fig. 1 depicts the c-automaton $\mathbb{C}ref$ used in [4, Introduction] as a running example. $\mathbb{C}ref$ specifies the coordination among participants \mathbb{C} , \mathbb{S} , and \mathbb{L} whereby a request **req** from client \mathbb{C} is served by server \mathbb{S} which replies with a message (of type) **res** and logs some meta-information **cnt** on a service \mathbb{L} (e.g., for billing purposes). Client \mathbb{C} may acknowledge a response of \mathbb{S} (i) with an **ok** message to restart the protocol, or (ii) by requiring a refinement of the response with a **ref** message, or else (iii) by ending the protocol with a **bye** message which \mathbb{S} forwards to \mathbb{L} . In the second case, \mathbb{S} sends \mathbb{C} either a **noRef** message, if no refinement is possible, or another **res** (with the corresponding **cnt** to \mathbb{L}). Using **Corinne** we can generate the projections of $\mathbb{C}ref$. Figure 2 contrasts the projection on participant \mathbb{S} returned by **Corinne** and the one in [4, Example 3.4], manually computed. The two CFSMs differ on the labels of the transition

² Determinisation required for projection is computed using the classical subset construction for FSAs with ϵ -transitions.

from state 4 to 6 and $\{5\}$ to $\{6\}$, respectively from the left and the right CFSM, which should correspond to each other. In fact, the label **SC!bye** on the transition from $\{5\}$ to $\{6\}$ is wrong.

We can also check the well-formedness of $\mathbb{C}ref$, which is the conjunction of two conditions, well-branchedness and well-sequencedness. Intuitively, well-branchedness requires that all the participants are aware of which branch is taken in a choice, if they have to behave differently on the available branches. Well-sequencedness instead requires concurrency (due to communications involving disjoint sets of participants) to be explicitly represented as commuting diamonds. As expected **Corinne** reports $\mathbb{C}ref$ to be well-sequenced. Unexpectedly, the check of well-branchedness fails. This is shown in the message in the bottom part of Fig. 1. The message means that the third condition in [4, Definition 4.6] fails on the pair of paths 3-0-1-2-3 and 3-4-3 because of participant **L**. The reason is that **L** occurs in the former but not in the latter. This implies that, in case the system reaches state 3 and participant **C** keeps on choosing indefinitely to send message **ref** to **S**, participant **L** will never be aware of what is going on. So **L** gets stuck waiting for a message **bye** that will never arrive. We refer to [4] for further details on well-formedness conditions. We conjecture that $\mathbb{C}ref$ is nevertheless well-behaved under suitable fairness assumptions, but the theory in [4] needs to be generalised to prove it.

We now demonstrate the composition operation relying on the running example of [3], where (referring to the UML representations) the diagrams in [3, Fig. 5] and [3, Fig. 6] are composed to derive the one in [3, Fig. 8]. Figure 3 shows the two c-automata involved and the result of the composition. The top-left c-automaton (let's dub it CA1) represents the global behaviour of a system with participants **P**, **Q**, and **H** interacting according to the following protocol. Participant **P** keeps on sending text messages to **Q**, which has to deliver them to **H**. Participant **P** can send a new message only if **H** has ascertained the propriety of language of the previous one, i.e. if the latter does not contain, say, rude or offensive words. Participant **H** acknowledges to **Q** the propriety of language of a received text by means of the message **ack**. In such a case, **Q** sends to **P** an **ok** message so that **P** can proceed by sending a further message. If the message does not pass the check, then **H** sends a **nack** message to inform **Q** that the text has not the required propriety of language. In such a case, **Q** produces **transf** (a semantically invariant reformulation of the text), sends it back to **H** and so that it can be checked. Before doing that, **Q** informs **P** (through the **notyet** message) that the text has not been accepted yet and a reformulation has been requested. After receiving a message, **H** may also decide to stop the interaction, sending a **stop** message to inform **Q** that no more text will be accepted. In such a case, **Q** informs **P** of that.

The bottom-left c-automaton (let's dub it CA2), instead describes a system formed by participants **K**, **R**, and **S** interacting according to the following protocol. Participant **K** sends text messages to **R** and **S** in an alternating way, starting with **R**. Participants **R** and **S** inform **K** that a text has been accepted or refused by sending back, respectively, either **ack** or **nack**. In the former case, it is the

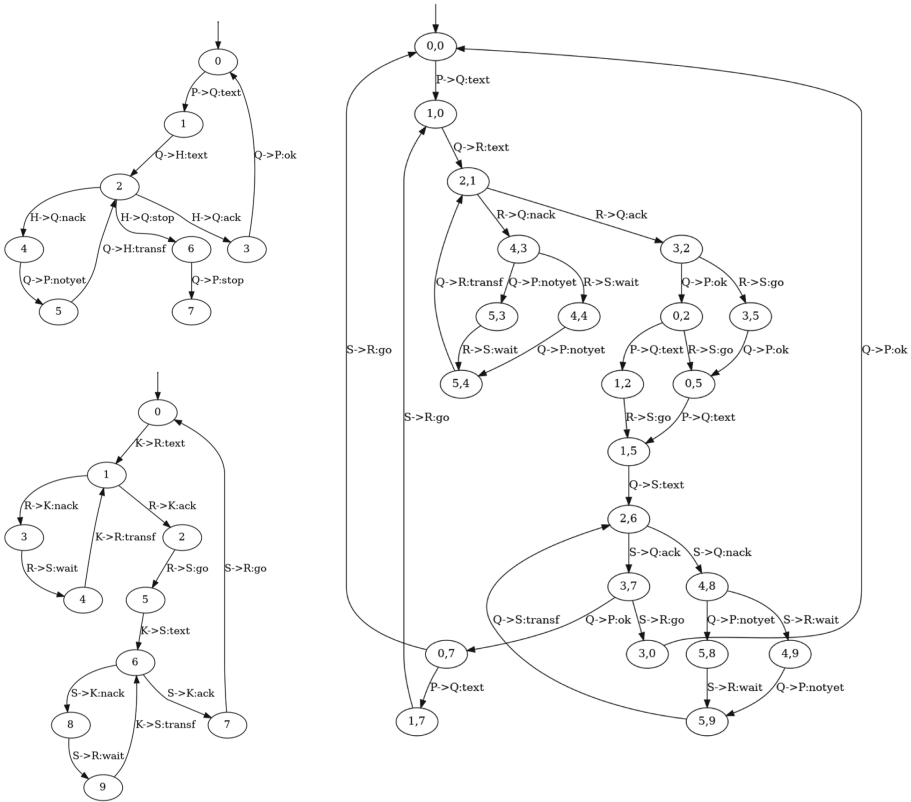


Fig. 3. Composition of c-automata in *Corinne*

other receiver’s turn to receive the text: a message **go** is exchanged between **R** and **S** to signal this case. In case **nack** is sent back, the sender has to resend the text until it is accepted. Meanwhile, the participant currently selected by **K** asks the other one to **wait**, since the previous message is being resent in a **transformed** form.

In the composition, participants **H** and **K** in, respectively, $\mathbb{CA}1$ and $\mathbb{CA}2$ of Fig. 3 are chosen as interfaces. This means that, e.g., when participant **Q** sends a **text**, it will send it alternatively to **R** and **S**. This can be thought somehow as if $\mathbb{CA}1$ invokes $\mathbb{CA}2$ for sending the message. However, w.r.t. choreographies with procedure invocations such as [19,20], our approach on the one hand allows a complex interaction between caller and callee choreographies but, on the other hand, does not allow for parameter passing in the invocation. Notice that the interfaces **H** and **K** are *compatible*; specifically, the languages of $\mathbb{CA}1 \downarrow_H$ and $\mathbb{CA}2 \downarrow_K$ are *dual* to each other if we disregard the name of participants other than **H** and **K** in the input/output actions (duality corresponds to the exchange of ‘!’ with ‘?’ and vice versa in the actions). Compatibility, roughly, enables the

composition not to modify in an essential way the behaviour of participants other than H and K . To obtain in *Corinne* the composition of $CA1$ and $CA2$ via the chosen interfaces H and K , we need first to apply *Product* on the two c -automata and then to apply *Synch* on H and K on the result. The only relevant difference between the composition performed by *Corinne* and the one in [3] is that the synchronisation in [3] transforms H and K into gateways while our composition drops them. Actually, our composition precisely corresponds to the direct composition in [3], but there is no example in [3] of this form of composition. We can obtain our result from the one in [3] by transforming sequences of communications $A-H-K-B$ into $A-B$ and $B-K-H-A$ into $B-A$ for any A and B . We also remark that the representation as c -automata highlights concurrency as commuting diamonds, e.g., the one at states $(4, 3)$, $(5, 3)$, $(4, 4)$, $(5, 4)$. Both the component c -automata and their composition can be checked to be both well-sequenced and well-branched. However, the check of well-branchedness on the composition is quite heavy.

4 Conclusion, Related Work, and Future Work

We refer to [4] for a comparison between c -automata and related models, while here we focus on the relations between *Corinne* and the most related tools.

Possibly *Corinne*'s closest sibling is *ChorGram*, a tool chain based on global graphs to support choreographic development of message-oriented applications [17, 29]. Global graphs are not directly comparable with c -automata: on the one hand they are more general since they allow one to specify parallelism, but on the other hand they require structured interactions. As a result, global graphs without parallel composition correspond to a strict subset of c -automata. In a sense, *ChorGram* complements *Corinne*'s functionalities; for instance, it supports different semantics of global graphs and some experimental ideas on choreography amendment or model-driven testing of message passing applications [18]. While *Corinne* can already take as input global graphs without parallel composition produced by *ChorGram*, we plan to further integrate the two tools in the future. We are currently considering to encode the parallel composition of global graphs as interleaving of independent transitions. We also plan to extend *ChorGram* so that it imports c -automata produced by *Corinne*. This paves the way to extensions of *Corinne* with features to import models based, e.g., on multiparty session types such as [21, 33] once proper mappings to c -automata are defined. We remark that this might not be simple for models relying on asynchronous communications such as [13, 30] for *Corinne*'s semantics is synchronous.

Another toolkit close to *Corinne* is *CAT*, a tool introduced in [7] to support the verification of communication protocols expressed as contract automata via the analysis of *agreement* properties. Contract automata are a versatile model of automata featuring the synthesis of controllers for communicating components; a thorough analysis based on *CAT* of the relations between choreographic and orchestration-based controllers (initiated in [8]) has been recently developed in [6]. This suggests a possible entanglement of the complementary features of

Corinne and **CAT** also in the light of the recent refactoring of the latter tool [9]. In fact, recently this model is being applied to choreography automata; **Corinne** could be useful in this context to validate choreography automata synthesised with contract automata.

We believe that **Corinne** is useful to experiment with c-automata, yet a number of improvements are desirable. First, right now the complexity of the check for well-branchedness is too high. We believe this can be reduced, at least in the average case, by avoiding checking multiple times analogous choices which are repeated in many states due to concurrency. Also, other functionalities would be useful, such as performing composition via gateways as described in [3] or checking well-formedness conditions also for the asynchronous semantics [4].

References

1. Barbanera, F., de'Liguoro, U., Hennicker, R.: Global types for open systems. In: Bartoletti, M., Knight, S. (eds.) ICE, EPTCS, vol. 279, pp. 4–20 (2018)
2. Barbanera, F., de'Liguoro, U., Hennicker, R.: Connecting open systems of communicating finite state machines. *J. Logic Algebr. Methods Program.* **109** (2019). <https://doi.org/10.1016/j.jlamp.2019.07.004>. Extended version of [1]
3. Barbanera, F., Dezani-Ciancaglini, M., Lanese, I., Tuosto, E.: Composition and decomposition of multiparty sessions. *J. Logic Algebr. Methods Program.* **119**, 100620 (2021)
4. Barbanera, F., Lanese, I., Tuosto, E.: Choreography automata. In: Bliudze, S., Bocchi, L. (eds.) COORDINATION 2020. LNCS, vol. 12134, pp. 86–106. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50029-0_6
5. Barbanera, F., Lanese, I., Tuosto, E.: Composition of choreography automata. Technical reports 2107.06727, Arxiv, July 2021. <http://arxiv.org/abs/2107.06727>
6. Basile, D., ter Beek, M.H., Pugliese, R.: Synthesis of orchestrations and choreographies: bridging the gap between supervisory control and coordination of services. *Logic. Methods Comput. Sci.* **16**(2) (2020)
7. Basile, D., Degano, P., Ferrari, G.-L., Tuosto, E.: Playing with our CAT and communication-centric applications. In: Albert, E., Lanese, I. (eds.) FORTE 2016. LNCS, vol. 9688, pp. 62–73. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39570-8_5
8. Basile, D., Degano, P., Ferrari, G., Tuosto, E.: Relating two automata-based models of orchestration and choreography. *J. Logic Algebr. Methods Program.* **85**(3), 425–446 (2016)
9. Basile, D., ter Beek, M.H.: A clean and efficient implementation of choreography synthesis for behavioural contracts. In: Damiani, F., Dardha, O. (eds.) COORDINATION 2021. LNCS, vol. 12717, pp. 225–238. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78142-2_14
10. Brand, D., Zafropulo, P.: On communicating finite-state machines. *J. ACM* **30**(2), 323–342 (1983)
11. Bravetti, M., Zavattaro, G.: Towards a unifying theory for choreography conformance and contract compliance. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 34–50. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77351-1_4



12. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_2
13. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: POPL, pp. 263–274 (2013). <https://doi.org/10.1145/2429069.2429101>
14. Corinne github repository. <https://github.com/lanese/corinne-3>
15. Coto, A., Guanciale, R., Lange, J., Tuosto, E.: ChorGram: tool support for choreographic development (2015). https://bitbucket.org/emlio_tuosto/chorgram/wiki/Home
16. Coto, A., Guanciale, R., Tuosto, E.: An abstract framework for choreographic testing. In: Lange, J., Mavridou, A., Safina, L., Scalas, A. (eds.) Proceedings 13th Interaction and Concurrency Experience, ICE 2020, Online, 19 June 2020. EPTCS, vol. 324, pp. 43–60 (2020)
17. Coto, A., Guanciale, R., Tuosto, E.: Choreographic development of message-passing applications. In: Bludze, S., Bocchi, L. (eds.) COORDINATION 2020. LNCS, vol. 12134, pp. 20–36. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50029-0_2
18. Coto, A., Guanciale, R., Tuosto, E.: An abstract framework for choreographic testing. *J. Logic Algebraic Methods Program.* **123**, 100712 (2021). Extended version of [16]
19. Cruz-Filipe, L., Montesi, F.: Procedural choreographic programming. In: Bouajjani, A., Silva, A. (eds.) FORTE 2017. LNCS, vol. 10321, pp. 92–107. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60225-7_7
20. Demangeon, R., Honda, K.: Nested protocols in session types. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 272–286. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32940-1_20
21. Dezani-Ciancaglini, M., Ghilezan, S., Jaksic, S., Pantovic, J., Yoshida, N.: Precise subtyping for synchronous multiparty sessions. In: Gay, S., Alglave, J. (eds.) Proceedings Eighth International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2015, London, UK, 18th April 2015. EPTCS, vol. 203, pp. 29–43 (2015). <https://doi.org/10.4204/EPTCS.203.3>
22. Domitilla github repository. <https://github.com/dedo94/Domitilla>
23. The DOT Language. <https://graphviz.org/doc/info/lang.html>
24. Fu, X., Bultan, T., Su, J.: Conversation protocols: a formalism for specification and verification of reactive electronic services. *Theoret. Comput. Sci.* **328**(1–2), 19–37 (2004). <https://doi.org/10.1016/tcs.2004.07.004>
25. Graphviz 0.16 - Simple Python interface for Graphviz. <https://pypi.org/project/graphviz/>
26. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Necula, G.C., Wadler, P. (eds.) POPL, pp. 273–284. ACM Press (2008)
27. Hüttel, H., et al.: Foundations of session types and behavioural contracts. *ACM Comput. Surv.* **49**(1), 3:1–3:36 (2016)
28. Kavantzias, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y., Barreto, C.: Web services choreography description language version 1.0. Technical report, W3C (2005). <http://www.w3.org/TR/ws-cdl-10/>
29. Lange, J., Tuosto, E., Yoshida, N.: A tool for choreography-based analysis of message-passing software. In: Gay, S., Ravara, A. (eds.) Behavioural Types: From Theory to Tools, chap. 6, pp. 125–146. Automation, Control and Robotics, River (2017)

30. Ng, N., Yoshida, N., Honda, K.: Multiparty session c: safe parallel programming with message optimisation. In: Furia, C.A., Nanz, S. (eds.) TOOLS 2012. LNCS, vol. 7304, pp. 202–218. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30561-0_15
31. OMG: Business Process Model and Notation (BPMN), Version 2.0, January 2011. <https://www.omg.org/spec/BPMN>
32. Parr, T.: Antlr. <https://www.antlr.org/index.html>
33. Severi, P., Dezani-Ciancaglini, M.: Observational equivalence for multiparty sessions. *Fundam. Informaticae* **170**(1–3), 267–305 (2019). <https://doi.org/10.3233/FI-2019-1863>
34. TKinter - Python interface to Tcl/Tk. <https://docs.python.org/3/library/tkinter.html>
35. Tuosto, E., Guanciale, R.: Semantics of global view of choreographies. *J. Logic Algebr. Methods Program.* **95**, 17–40 (2018)

Verification



Specification and Safety Verification of Parametric Hierarchical Distributed Systems

Marius Bozga  and Radu Iosif ^(✉) 

Univ. Grenoble Alpes, CNRS, Grenoble INP (Institute of Engineering
Univ. Grenoble Alpes), VERIMAG, 38000 Grenoble, France
{marius.bozga,radu.iosif}@univ-grenoble-alpes.fr

Abstract. We introduce a term algebra as a new formal specification language for the coordinating architectures of distributed systems consisting of a finite yet unbounded number of components. The language allows to describe infinite sets of systems whose coordination between components share the same pattern, using inductive definitions similar to the ones used to describe algebraic data types or recursive data structures. Further, we give a verification method for the parametric systems described in this language, relying on the automatic synthesis of structural invariants that enable proving general safety properties (absence of deadlocks and critical section violations). The invariants are defined using the $WS\kappa S$ fragment of the monadic second order logic, known to be decidable by a classical automata-logic connection. This reduces the safety verification problem to checking satisfiability of a $WS\kappa S$ formula. We implemented the invariant synthesis method into a prototype tool and carried out verification experiments on a number of non-trivial models specified using the term algebra.

1 Introduction

The separation between behavior and coordination is a fundamental principle in the design of large-scale distributed systems [16]. By *behavior* we mean a set of traces of observable events. A *component* is a representation of a behavior, by means of a (finite) state machine, whose actions are labeled by events. The *architecture* of the system defines the interactions between components, as sets of events that must occur simultaneously in several components. For instance, Fig. 1a shows a token-ring systems, whose components are depicted in yellow boxes (behaviors are modeled by the finite-state machines within the boxes) and whose architecture is the set of connections between components (depicted with solid lines). Such high-level models of real-life distributed systems are suitable for reasoning about correctness in the early stages of system design, when details related to network reliability or the implementation of coordination mechanisms, by means of low-level synchronization mechanisms (e.g. semaphores, monitors, compare-and-swap, etc.) are abstracted away.

This modular view of a distributed system is key to scalable design methods that exploit a conceptual hierarchy, in which each module is split into sub-modules. For instance, a ring is a chain whose final output port is connected to the initial input port, whereas a chain consists of a (head) component linked to a separate (tail) chain (Fig. 1b). Furthermore, system designers are accustomed to the use of predefined *architectural patterns*, that define the interactions between (unboundedly large) sets of modules (e.g. crowds, rings, pipelines, stars, trees, etc.). In this context, the contribution of the paper is three-fold.

1. We introduce a formal language to describe the coordinating architectures of distributed systems parameterized by (i) the number of components of each type that are active in the system, e.g. a system with n readers and m writers, in which n and m are not known a priori and (ii) the pattern in which the interactions occur (e.g. a pipeline, ring, star or more general hypergraph structures). The language uses predicate symbols to hierarchically break the architecture into sub-modules. The interpretation of these predicate symbols is defined inductively by rewriting rules consisting of terms that contain predicate atoms, in a way that recalls the usual definitions of algebraic datatypes [2] or heaps [18].
2. We tackle the *parametric safety problem* for systems described in this language, which is checking that the reachable states of every instance stays clear of a set of global error configurations, such as deadlocks or critical section violations. We synthesize invariants directly from the syntactic description of the system, generate WS κ S formulæ [19] that are unsatisfiable only if every system described by the given inductive definitions is safe and use off-the-shelf WS κ S solvers [11] for proving safety automatically. The invariant synthesis method models the set of executions of a parametric system as a boolean (1-safe) Petri net of unbounded size and computes structural invariants (trap invariants, linear invariants) of this Petri net.
3. We implemented the invariant synthesis in a prototype tool and experimented with a number of parametric component-based systems with non-trivial architectural patterns, such as trees with root links, trees with linked leaves, token-rings with or without a main controller, etc.

Example 1. Let us consider a distributed system consisting of components of type C , having two interaction ports, namely *in* and *out* and whose behavior is described by a finite state machine with transitions $q_0 \xrightarrow{out} q_1$ and $q_1 \xrightarrow{in} q_0$. These components are arranged in a ring, such that the *out* port of a component is connected to the *in* port of its right neighbour, with the exception of the last component, whose *out* port connects to the *in* port of the first component (Fig. 1a). The connections (interactions) in the system are described by the predicate $Ring()$, defined inductively by the rules below:

$$Ring() \leftarrow \nu y_1 \nu y_2 . \langle out(y_2) \cdot in(y_1) \rangle (Chain(y_1, y_2)) \quad (1)$$

$$Chain(x_1, x_2) \leftarrow \langle out(x_1) \cdot in(x_2) \rangle (Comp(x_1), Comp(x_2)) \quad (2)$$

$$Chain(x_1, x_2) \leftarrow \nu y_1 . \langle out(x_1) \cdot in(y_1) \rangle (Comp(x_1), Chain(y_1, x_2)) \quad (3)$$

$$Comp(x) \leftarrow C(x) \quad (4)$$

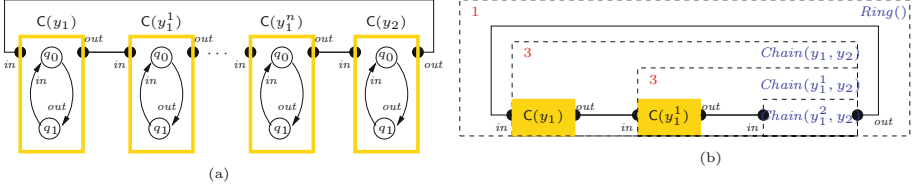


Fig. 1. Recursive Specification of a Token-Ring System (Color figure online)

Rule (1) states that a $Ring()$ consist of a $Chain(y_1, y_2)$, where y_1 and y_2 are the indices of the first and last components¹, respectively. The last out port is connected to the first in port, written as $out(y_2) \cdot in(y_1)$. Rule (2) states that the least $Chain(x_1, x_2)$ consists of two instances of type C , namely $C(x_1)$ and $C(x_2)$, and the out port of x_1 connects to the in port of x_2 , described as $out(x_1) \cdot in(x_2)$. Rule (3) gives the inductive step, namely that every $Chain(x_1, x_2)$ consists of a component $C(x_1)$ that interacts with a disjoint chain from y_1 to x_2 . Here the binder νy_1 makes sure the value of y_1 is different from the value of every other variable in the system. Since this binder is used in a recursive rule, each identifier in a subsequent unfolding of $Chain(y_1, x_2)$ is guaranteed to be unique. Last, rule (4) is used to instantiate (i.e. create new) components of type C . In principle, this rule is not necessary, as any occurrence of a predicate $Comp(y)$ can be replaced by a component $C(y)$, however it is considered for technical reasons related to our invariant-based verification approach.

Figure 1b shows the unfoldings of this set of recursive definitions. The system depicted in Fig. 1a is obtained by an application of rule (1), followed by n applications of rule (3), ending with an application of rule (2). The first two applications of (3) following the application of (1) are depicted in Fig. 1b, with rule labels annotated (each application of rule (3) creates a fresh variable, denoted by y_1^1, \dots, y_1^n , respectively). ■

2 Preliminaries

This section introduces preliminary definitions used throughout the paper. Given sets A and B , we denote by $A \rightarrow B$ the set of total functions $f : A \rightarrow B$. If $\mathbf{a} = \langle a_1, \dots, a_n \rangle$ is a tuple of values from A , then $f(\mathbf{a}) \stackrel{\text{def}}{=} \langle f(a_1), \dots, f(a_n) \rangle$. By $\mathbf{a} \cdot \mathbf{b}$ we denote the concatenation of tuples \mathbf{a} and \mathbf{b} . For two positive integers $k, \ell \in \mathbb{N}$, we denote by $[k, \ell]$ the set $\{k, k + 1, \dots, \ell\}$, assumed to be empty if $k > \ell$. The cardinality of a finite set A is denoted by $|A|$.

Trees. Trees play a key role in the definition of parametric distributed systems from the following section (Sect. 3). Let $\kappa \geq 1$ be an integer constant, fixed throughout this paper, and let $[1, \kappa]^*$ denote the set of finite sequences of integers

¹ First and last are understood here in the order of unfolding of the rewriting rules.

between 1 and κ , called *nodes* in the following. A κ -ary tree \mathcal{T} is a partial function mapping $[1, \kappa]^*$ to a set of labels. The domain of \mathcal{T} , denoted $\text{nodes}(\mathcal{T})$, is such that $wi \in \text{nodes}(\mathcal{T})$ for some $i \in [1, \kappa]$ only if $w \in \text{nodes}(\mathcal{T})$ and $wj \in \text{nodes}(\mathcal{T})$ for all $j \in [1, i-1]$. The *root* of \mathcal{T} is the empty sequence ϵ , the *children* of a node $w \in \text{nodes}(\mathcal{T})$ are $\{wi \in \text{nodes}(\mathcal{T}) \mid i \in [1, \kappa]\}$ and the *parent* of a node wi , $i \in [1, \kappa]$, is w . The *leaves* of \mathcal{T} are $\text{leaves}(\mathcal{T}) \stackrel{\text{def}}{=} \{w \in \text{nodes}(\mathcal{T}) \mid w.1 \notin \text{nodes}(\mathcal{T})\}$. The *subtree* of \mathcal{T} rooted at w is defined by $\text{nodes}(\mathcal{T}\downarrow_w) \stackrel{\text{def}}{=} \{w' \mid ww' \in \text{nodes}(\mathcal{T})\}$ and $\mathcal{T}\downarrow_w(w') \stackrel{\text{def}}{=} \mathcal{T}(ww')$, for all $w' \in \text{nodes}(\mathcal{T}\downarrow_w)$.

The invariant synthesis method uses the restriction of monadic second order logic to trees of branching degree κ and quantification over finite sets only. Let $\mathbb{V}_1 = \{x, y, z, \dots\}$ and $\mathbb{V}_2 = \{X, Y, Z, \dots\}$ be countably infinite sets of first and second order variables, respectively. The formulæ of the WS κ S logic are defined inductively by the syntax:

$$\begin{aligned} \tau &::= \epsilon \mid x \in \mathbb{V}_1 \mid \tau_1.i, \quad i \in [1, \kappa] && \text{terms} \\ \phi &::= \tau = \tau \mid X(\tau) \mid \phi \wedge \phi \mid \neg\phi \mid \exists x . \phi \mid \exists X . \phi && \text{formulæ} \end{aligned}$$

As usual, we write $\phi_1 \vee \phi_2 \stackrel{\text{def}}{=} \neg(\neg\phi_1 \wedge \neg\phi_2)$, $\phi_1 \rightarrow \phi_2 \stackrel{\text{def}}{=} \neg\phi_1 \vee \phi_2$, $\phi_1 \leftrightarrow \phi_2 \stackrel{\text{def}}{=} (\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$, $\forall x . \phi \stackrel{\text{def}}{=} \neg\exists x . \neg\phi$ and $\forall X . \phi \stackrel{\text{def}}{=} \neg\exists X . \neg\phi$. WS κ S formulæ are interpreted over an infinite tree, where first order variables $x \in \mathbb{V}_1$ range over individual nodes $n \in [1, \kappa]^*$, second order variables $X \in \mathbb{V}_2$ range over *finite sets* of nodes, ϵ is a constant symbol interpreted as the root of the tree and, for all $i \in [1, \kappa]$, the notation $.i$ is interpreted as the function mapping each $w \in [1, \kappa]^*$ into wi . Given a valuation $\nu : \mathbb{V}_1 \cup \mathbb{V}_2 \rightarrow [1, \kappa]^* \cup 2^{[1, \kappa]^*}$, such that $\nu(x) \in [1, \kappa]^*$, for each $x \in \mathbb{V}_1$ and $\nu(X) \subseteq [1, \kappa]^*$ ($\nu(X)$ is finite), for each $X \in \mathbb{V}_2$, the satisfaction relation \models is defined inductively, as usual [14]. A valuation ν is a *model* of a formula ϕ iff $\nu \models \phi$. A formula is *satisfiable* if and only if it has a model.

Petri Nets. A *Petri net* (PN) is a tuple $\mathbf{N} = \langle S, T, E \rangle$, where S is a set of *places*, T is a set of *transitions*, $S \cap T = \emptyset$, and $E \subseteq (S \times T) \cup (T \times S)$ is a set of *edges*. Given $x, y \in S \cup T$, we write $E(x, y) \stackrel{\text{def}}{=} 1$ if $(x, y) \in E$ and $E(x, y) \stackrel{\text{def}}{=} 0$, otherwise. Let $\bullet x \stackrel{\text{def}}{=} \{y \in S \cup T \mid E(y, x) = 1\}$, $x^\bullet \stackrel{\text{def}}{=} \{y \in S \cup T \mid E(x, y) = 1\}$ and lift these definitions to sets of nodes. A *marking* of \mathbf{N} is a function $m : S \rightarrow \mathbb{N}$. A transition t is *enabled* in m if and only if $m(s) > 0$ for each place $s \in \bullet t$. We write $m \xrightarrow{t} m'$ whenever t is enabled in m and $m'(s) = m(s) - E(s, t) + E(t, s)$, for all $s \in S$ and $t \in T$. A sequence of transitions $\sigma = t_1, \dots, t_n$ is a *firing sequence*, written $m \xrightarrow{\sigma} m'$ if and only if either (i) $n = 0$ and $m = m'$, or (ii) $n \geq 1$ and there exist markings m_1, \dots, m_{n-1} such that $m \xrightarrow{t_1} m_1 \dots m_{n-1} \xrightarrow{t_n} m'$.

A *marked Petri net* is a pair $\mathcal{N} = (\mathbf{N}, m_0)$, where m_0 is the *initial marking* of \mathbf{N} . A marking m is *reachable* in \mathcal{N} if there exists a firing sequence σ such that $m_0 \xrightarrow{\sigma} m$. We denote by $\mathcal{R}(\mathcal{N})$ the set of reachable markings of \mathcal{N} . A marked PN \mathcal{N} is *boolean* if $m(s) \leq 1$, for each $s \in S$ and $m \in \mathcal{R}(\mathcal{N})$. All marked PNs considered in the following will be boolean and we shall silently blur the distinction between a marking $m : S \rightarrow \{0, 1\}$ and the set $\{s \in S \mid m(s) = 1\}$.

Given a set of markings \mathcal{E} , a marked PN \mathcal{N} is *safe w.r.t.* \mathcal{E} if and only if $\mathcal{R}(\mathcal{N}) \cap \mathcal{E} = \emptyset$. A set of markings \mathcal{M} is an *inductive invariant* of $\mathcal{N} = (\mathbf{N}, m_0)$ if

and only if $m_0 \in \mathcal{M}$ and for each $m \xrightarrow{t} m'$ such that $m \in \mathcal{M}$, we have $m' \in \mathcal{M}$. It is known that $\mathcal{R}(\mathcal{N})$ is the least inductive invariant of \mathcal{N} , thus \mathcal{N} is safe w.r.t \mathcal{E} if it has an inductive invariant \mathcal{M} disjoint from \mathcal{E} .

Components. In this paper we are concerned with systems consisting of an unbounded number of components that are replicas of a fairly small set of patterns, called component types. Let $\mathbb{P} = \{a, b, \dots\}$ and $\mathbb{S} = \{s, t, \dots\}$ be countably infinite sets of *ports* and *states*, respectively. An injective function P (resp. S) mapping tree nodes to ports (resp. states) is called a *port type* (resp. *state type*). A *component type* is a tuple $\mathcal{B} = \langle \mathcal{P}, \mathcal{S}, \mathcal{I}, \Delta \rangle$, where $\mathcal{P} \subseteq [1, \kappa]^* \rightarrow \mathbb{P}$ and $\mathcal{S} \subseteq [1, \kappa]^* \rightarrow \mathbb{S}$ are finite sets of port and state types, $\mathcal{I} \in \mathcal{S}$ is the *initial* state type, and Δ is a finite set of *transition rules* $S \xrightarrow{P} T$, where $S, T \in \mathcal{S}$ and $P \in \mathcal{P}$. In addition, we require that (i) the elements of \mathcal{P} (resp. \mathcal{S}) have pairwise disjoint ranges and (ii) for any two transition rules $S_1 \xrightarrow{P_1} S'_1, S_2 \xrightarrow{P_2} S'_2$, if $P_1 = P_2$ then $S_1 = S_2$ and $S'_1 = S'_2$. For a transition rule $S \xrightarrow{P} S' \in \Delta$, let $\bullet P \stackrel{\text{def}}{=} S$ and $P \bullet \stackrel{\text{def}}{=} S'$ denote the pre- and post-state type of the unique transition rule whose label is the port type P .

The replicas of a component type are indexed (distinguished) by tree nodes². Given a component type $\mathcal{B} = \langle \mathcal{P}, \mathcal{S}, \mathcal{I}, \Delta \rangle$ and a tree node $w \in [1, \kappa]^*$, we define the *component* $\mathcal{B}(w) \stackrel{\text{def}}{=} \langle \{P(w) \mid P \in \mathcal{P}\}, \{S(w) \mid S \in \mathcal{S}\}, \mathcal{I}(w), \{S(w) \xrightarrow{P(w)} S'(w) \mid S \xrightarrow{P} S' \in \Delta\} \rangle$. Note that the sets of ports $\{P(w) \mid P \in \mathcal{P}\}$ (resp. states $\{S(w) \mid S \in \mathcal{S}\}$) of different replicas of the same component type are disjoint, because the port (state) types are required to have disjoint ranges. We slightly abuse notation by writing $\bullet(P(w)) \stackrel{\text{def}}{=} \bullet(P)(w)$ and $(P(w)) \bullet \stackrel{\text{def}}{=} (P) \bullet(w)$ (we omit brackets when they are clear from the context). We consider below a set \mathbb{B} of component types, with pairwise disjoint sets of port and state types.

Architectures. The coordination in a system is defined by architectures. An *interaction* $\pi \in 2^{\mathbb{P}}$ is a finite set of ports. An *architecture* $\gamma \subseteq 2^{\mathbb{P}}$ is a finite set of interactions. Given component types $\mathcal{B}_i = \langle \mathcal{P}_i, \mathcal{S}_i, \mathcal{I}_i, \Delta_i \rangle \in \mathbb{B}$ and tree nodes $w_i \in [1, \kappa]^*$, the *behavior* of the system consisting of the components $\mathcal{B}_i(w_i)$, $i = 1, \dots, n$, coordinated by the architecture γ is defined by the marked PN $\gamma(\mathcal{B}_1(w_1), \dots, \mathcal{B}_n(w_n)) \stackrel{\text{def}}{=} (\langle S, \gamma, E \rangle, m_0)$, where $S \stackrel{\text{def}}{=} \bigcup_{i=1}^n \{S(w_i) \mid S \in \mathcal{S}_i\}$ is the set of places, for each interaction $\pi \in \gamma$, the edges to (from) π are given by $\bullet \pi \stackrel{\text{def}}{=} \{\bullet p \mid p \in \pi\}$ ($\pi \bullet \stackrel{\text{def}}{=} \{p \bullet \mid p \in \pi\}$) and the initial marking is $m_0 \stackrel{\text{def}}{=} \{\mathcal{I}_i(w_i) \mid i \in [1, n]\}$.

Example 2. Figure 2 shows the marked PN that defines the behavior of the system from Fig. 1a. The tree node $1 \dots 1$ (i times) is represented by its value i in the unary encoding. The interaction $\{in(n), out(1)\}$ is duplicated, for readability. The initially marked places are surrounded by dashed circles. ■

² We identify components by tree nodes in preparation of the ground for the verification technique from Sect. 4. However, these definitions can be given in general, for any countably infinite set of identifiers.

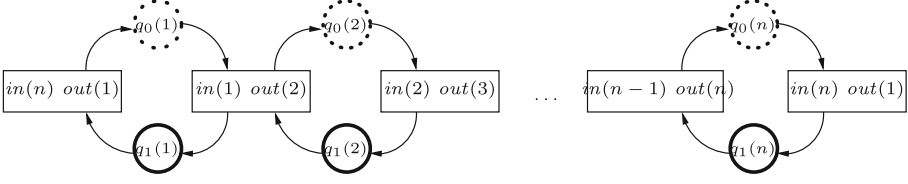


Fig. 2. The Behavior of a Token-Ring System

3 A Term Algebra of Behaviors

In this section we introduce a term algebra for describing the systems resulting from the application of an architecture to an unbounded number of component type instances (see Example 1 for the specification of a token-ring system in this language). Let \mathbb{A} be a countably infinite set of *predicate symbols*, and let $\#(\mathbf{A})$ denote the arity of the predicate symbol $\mathbf{A} \in \mathbb{A}$.

Syntax. The following syntax generates *behavioral terms* inductively:

$$\begin{array}{ll}
 P \in [1, \kappa]^* \mapsto \mathbb{P}, & x \in \mathbb{V}_1, \mathcal{B} \in \mathbb{B}, \mathbf{A} \in \mathbb{A} \\
 I ::= P(x) \mid I_1 \cdot I_2 & \text{interactions} \\
 \Gamma ::= I \mid \Gamma_1 + \Gamma_2 & \text{architectures} \\
 \mathbf{b} ::= \mathcal{B}(x) \mid \langle \Gamma \rangle(\mathbf{b}_1, \dots, \mathbf{b}_n) \mid \nu x . \mathbf{b}_1 \mid \mathbf{A}(x_1, \dots, x_{\#(\mathbf{A})}) & \text{behavioral terms}
 \end{array}$$

A variable x occurring in a behavioral term \mathbf{b} is *free* if it does not occur in the scope of some subterm of the form $\nu x . \mathbf{b}_1$ and *bound* otherwise. The set of free variables occurring in a term \mathbf{b} is denoted by $\text{fv}(\mathbf{b})$. In the following, we assume that all bound variables occurring in a term are pairwise distinct and distinct from the free variables. This assumption loses no generality because terms obtained by α -conversion (renaming of bound variables) are usually viewed as the same term. A term \mathbf{b} is *closed* if $\text{fv}(\mathbf{b}) = \emptyset$ and *predicateless* if no predicates from \mathbb{A} occur in \mathbf{b} . We denote by $\mathbf{b}[y_1/x_1, \dots, y_n/x_n]$ the term obtained by substituting the variable $x_i \in \text{fv}(\mathbf{b})$ with y_i , for each $i \in [1, n]$. We write $\text{size}(\mathbf{b})$ for the number of occurrences of symbols in \mathbf{b} .

A term $\mathcal{B}(x)$ is called an *instance atom* and a term $\mathbf{A}(x_1, \dots, x_n)$ is called a *predicate atom*. We denote by $\#\text{pred}(\mathbf{b})$ the number of occurrences of predicate atoms and by $\text{pred}_i(\mathbf{b})$ the predicate atom that occurs i -th in \mathbf{b} , for $i \in [1, \#\text{pred}(\mathbf{b})]$, in some linear order of the nodes in the syntax tree of \mathbf{b} . The predicate symbols are interpreted as the least sets of predicateless terms inductively defined by a rewriting system:

Definition 1. A rewriting system is a finite set \mathcal{R} of rules of one of the forms:

$$\begin{array}{l}
 \mathbf{A}(x) \leftarrow \mathcal{B}(x) \\
 \mathbf{A}(x_1, \dots, x_{\#(\mathbf{A})}) \leftarrow \nu y_1 \dots \nu y_n \cdot \langle \Gamma \rangle(\mathbf{A}_1(z_1^1, \dots, z_{\#(\mathbf{A}_1)}^1), \dots, \mathbf{A}_m(z_1^m, \dots, z_{\#(\mathbf{A}_m)}^m)) \\
 \text{where } m \geq 1 \text{ and } \left\{ \{z_1^i, \dots, z_{\#(\mathbf{A}_i)}^i\} \right\}_{i=1}^m \text{ is a partition of } \{x_1, \dots, x_{\#(\mathbf{A})}, y_1, \dots, y_n\}
 \end{array}$$

For instance, in Example 1, rule (4) is an instantiation rule, whereas (1), (2) and (3) are inductive rules. We write $A(x_1, \dots, x_{\#(A)}) \leftarrow_{\mathcal{R}} \mathbf{b}$ for $A(x_1, \dots, x_{\#(A)}) \leftarrow \mathbf{b} \in \mathcal{R}$. The *size* of \mathcal{R} is $\text{size}(\mathcal{R}) \stackrel{\text{def}}{=} \sum_{A(x_1, \dots, x_{\#(A)}) \leftarrow_{\mathcal{R}} \mathbf{b}} \text{size}(\mathbf{b})$. Given behavioral terms \mathbf{b}_1 and \mathbf{b}_2 , we denote by $\mathbf{b}_1 \stackrel{f}{\leftarrow} \mathbf{b}_2$ the rewriting step that obtains \mathbf{b}_2 by replacing a predicate atom $A(y_1, \dots, y_{\#(A)})$ in \mathbf{b}_1 with $\mathbf{b}[y_1/x_1, \dots, y_{\#(A)}/x_{\#(A)}]$, where $r = (A(x_1, \dots, x_{\#(A)}) \leftarrow \mathbf{b})$ is a rule of \mathcal{R} and all bound variables in \mathbf{b} are renamed to avoid clashes with the variables from \mathbf{b}_1 . We write $[\mathbf{b}]_{\mathcal{R}}$ for the set of predicateless terms obtained from \mathbf{b} by exhaustively applying the rewriting rules from \mathcal{R} to it.

Semantics. Let us consider a given closed behavioral term \mathbf{b} and a rewriting system \mathcal{R} . First, we define the semantics of a (closed) predicateless behavioral term $\mathbf{t} \in [\mathbf{b}]_{\mathcal{R}}$, as the behavior (i.e. marked PN) resulting from joining the components defined by the instance atoms from \mathbf{t} , via the architecture consisting of all the interactions that occur in \mathbf{t} . This definition is done in two steps:

- (a) we write \mathbf{t} in *prenex form* as $\nu x_1 \dots \nu x_n \cdot \mathbf{u}$, where \mathbf{u} contains no more terms of the form $\nu x \cdot \mathbf{b}$, by moving all the ν binders upfront. Because all bound variables in \mathbf{t} , including those introduced by rewriting, are assumed to be pairwise distinct, this step incurs no name clashes.
- (b) we apply the following *flattening* relation exhaustively:

$$\langle \Gamma_1 \rangle (\langle \Gamma_2 \rangle (\mathbf{b}_1, \dots, \mathbf{b}_i), \mathbf{b}_{i+1}, \dots, \mathbf{b}_n) \rightsquigarrow \langle \Gamma_1 + \Gamma_2 \rangle (\mathbf{b}_1, \dots, \mathbf{b}_n) \quad (5)$$

Example 3. Consider the below rewriting sequence, using rules from Example 1:

$$\begin{aligned} & \text{Ring}() \leftarrow \nu y_1 \nu y_2 \cdot \langle \text{out}(y_2) \cdot \text{in}(y_1) \rangle (\text{Chain}(y_1, y_2)) \\ & \leftarrow \nu y_1 \nu y_2 \cdot \langle \text{out}(y_2) \cdot \text{in}(y_1) \rangle (\nu y_1^1 \cdot \langle \text{out}(y_1) \cdot \text{in}(y_1^1) \rangle (\text{Comp}(y_1), \text{Chain}(y_1^1, y_2))) \\ & \leftarrow \nu y_1 \nu y_2 \cdot \langle \text{out}(y_2) \cdot \text{in}(y_1) \rangle (\nu y_1^1 \cdot \langle \text{out}(y_1) \cdot \text{in}(y_1^1) \rangle (\text{Comp}(y_1), \\ & \quad \langle \text{out}(y_1^1) \cdot \text{in}(y_2) \rangle (\text{Comp}(y_1^1), \text{Comp}(y_2)))) \\ & \leftarrow \dots \leftarrow \nu y_1 \nu y_2 \cdot \langle \text{out}(y_2) \cdot \text{in}(y_1) \rangle (\nu y_1^1 \cdot \langle \text{out}(y_1) \cdot \text{in}(y_1^1) \rangle (\text{C}(y_1), \\ & \quad \langle \text{out}(y_1^1) \cdot \text{in}(y_2) \rangle (\text{C}(y_1^1), \text{C}(y_2)))) = \mathbf{t} \end{aligned}$$

By applying flattening to the last term, we obtain:

$$\mathbf{t}^{\rightsquigarrow} = \nu y_1 \nu y_2 \cdot \langle \text{out}(y_2) \cdot \text{in}(y_1) + \text{out}(y_1) \cdot \text{in}(y_1^1) + \text{out}(y_1^1) \cdot \text{in}(y_2) \rangle (\text{C}(y_1), \text{C}(y_1^1), \text{C}(y_2))$$

■

Note that every chain $\mathbf{t}_1 \rightsquigarrow \mathbf{t}_2 \rightsquigarrow \dots$ is finite, because the height of terms strictly decreases with flattening. The result of flattening is of the form $\mathbf{t}^{\rightsquigarrow} \stackrel{\text{def}}{=} \langle \Gamma \rangle (\mathcal{B}_1(x_1), \dots, \mathcal{B}_n(x_n))$, where $\Gamma = \sum_{k=1}^m P_{k1}(x_{k1}) \cdot \dots \cdot P_{kr_k}(x_{kr_k})$ is an architecture description, such that each $P_{k\ell} \in [1, \kappa]^* \mapsto \mathbb{P}$ is a port type, $x_{k\ell} \in \{x_1, \dots, x_n\}$, for all $k \in [1, m]$ and $\ell \in [1, r_k]$. Given an injective valuation $\nu : \mathbb{V}_1 \rightarrow [1, \kappa]^*$ that maps variables into distinct nodes of a tree of branching degree κ , we define the architecture $\Gamma(\nu) \stackrel{\text{def}}{=} \{\{P_{k1}(\nu(x_{k1})), \dots, P_{kr_k}(\nu(x_{kr_k}))\} \mid k \in [1, m]\}$ and the behavior:

$$\mathbf{B}_{\nu}^{\mathbf{t}} \stackrel{\text{def}}{=} \Gamma(\nu)(\mathcal{B}_1(\nu(x_1)), \dots, \mathcal{B}_n(\nu(x_n))) \quad (6)$$

The semantics of the behavioral term \mathbf{b} in the rewriting system \mathcal{R} is the following set of marked PNs:

$$\llbracket \mathbf{b} \rrbracket_{\mathcal{R}} \stackrel{\text{def}}{=} \{ \mathbf{B}_{\nu}^{\mathbf{t}} \mid \mathbf{t} \in [\mathbf{b}]_{\mathcal{R}}, \nu \in \mathbb{V}_1 \rightarrow [1, \kappa]^* \text{ injective} \} \quad (7)$$

As a remark, the flattening step is required because applying an architecture to a set of components is a global operation; if an interaction $P_{k_1}(x_{k_1}) \cdot \dots \cdot P_{k_{r_k}}(x_{k_{r_k}})$ occurs as a monomial in the architecture description Γ of a subterm $\mathbf{u} = \langle \Gamma \rangle(\mathbf{t}_1, \dots, \mathbf{t}_{\ell})$ of \mathbf{t} and some variable x_{k_i} occurs in an instance atom $\mathcal{B}(x_{k_i})$ in \mathbf{t} but not in \mathbf{u} , the interaction would be ignored if we applied $\Gamma(\nu)$ directly to $\mathbf{B}_{\nu}^{\mathbf{t}_1}, \dots, \mathbf{B}_{\nu}^{\mathbf{t}_{\ell}}$, for some injective valuation ν .

4 The Parametric Safety Problem

Having defined a rewriting-based term algebra for the specification of distributed systems, we move on to the problem of verifying that every behavior generated by a given rewriting system \mathcal{R} , starting from a given behavioral term \mathbf{b} is safe with respect to a given set of error markings. This problem is challenging, because we ask for a proof of safety that holds for the behavior(s) of *every* predicateless rewriting of the behavioral term, i.e. for each $\mathbf{t} \in [\mathbf{b}]_{\mathcal{R}}$. Since, even for token-ring systems with finite-state components, the parametric safety problem is undecidable [9], we resort to a sound but necessarily incomplete solution, that consists in computing inductive invariants.

Structural Invariants. In contrast with the classical approach to invariant synthesis, based on a fixpoint iteration in an abstract domain [8], we consider a particular class of invariants, that can be obtained directly from the syntactic structure of the marked PN representation of behaviors. For this reason, we call these invariants *structural*. In the following, we define two kinds of such invariants, namely *trap* and *mutex* invariants:

Definition 2. *Given a marked PN $\mathcal{N} = (\langle S, T, E \rangle, \mathbf{m}_0)$, a set $\theta \subseteq S$ is a:*

1. trap if $|\theta \cap \mathbf{m}_0| \geq 1$ and, for any $t \in T$, if $|\theta \cap \bullet t| \geq 1$ then $|\theta \cap t \bullet| \geq 1$.
2. mutex if $|\theta \cap \mathbf{m}_0| = 1$ and, for any $t \in T$, we have $|\theta \cap \bullet t| = |\theta \cap t \bullet| \leq 1$.

The structural invariants of \mathcal{N} are the trap and mutex invariants, respectively:

- A. $\Theta(\mathcal{N}) \stackrel{\text{def}}{=} \{ \mathbf{m} \text{ marking of } \mathcal{N} \mid |\mathbf{m} \cap \theta| \geq 1, \text{ for each trap } \theta \text{ of } \mathcal{N} \}$
- B. $\Omega(\mathcal{N}) \stackrel{\text{def}}{=} \{ \mathbf{m} \text{ marking of } \mathcal{N} \mid |\mathbf{m} \cap \theta| = 1, \text{ for each mutex } \theta \text{ of } \mathcal{N} \}$.

Note that, since \mathcal{N} is boolean, each marking can be represented as a set of places. Moreover, it is easy to check that $\Theta(\mathcal{N})$ and $\Omega(\mathcal{N})$ contain the initial marking \mathbf{m}_0 and are closed under the transition relation of the net. Thus both sets are inductive invariants of \mathcal{N} , that can be used to prove a safety property, by checking the emptiness of the intersection of the above sets with a set \mathcal{E} of error markings.

Our method encodes the families of sets $\{ \Theta(\mathcal{N}) \mid \mathcal{N} \in \llbracket \mathbf{b} \rrbracket_{\mathcal{R}} \}$ and $\{ \Omega(\mathcal{N}) \mid \mathcal{N} \in \llbracket \mathbf{b} \rrbracket_{\mathcal{R}} \}$ by formulæ of WS κ S, for a suitable integer constant $\kappa \geq 1$. To

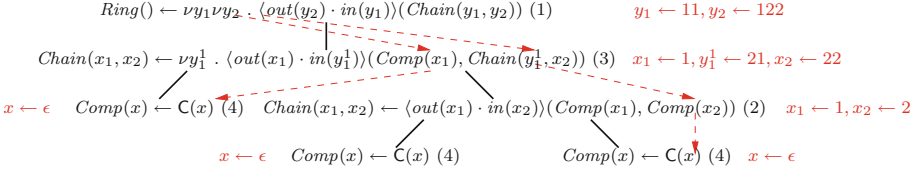


Fig. 3. A Rewriting Tree for the Token-Ring System

prove a parametric safety property given by a $\text{WS}\kappa\text{S}$ encoding of the \mathcal{E} set, a sufficient (but not necessary) condition is that the $\text{WS}\kappa\text{S}$ formula defining the family of sets $\{\Theta(\mathcal{N}) \cap \Omega(\mathcal{N}) \cap \mathcal{E} \mid \mathcal{N} \in \llbracket \mathbf{b} \rrbracket_{\mathcal{R}}\}$ is unsatisfiable. Since automata-theoretic decision procedures exist for $\text{WS}\kappa\text{S}$ [19], we rely on existing provers [11] to perform this check.

Rewriting Trees. The crux of the method is to represent each predicateless behavioral term $\mathbf{t} \in \llbracket \mathbf{b} \rrbracket_{\mathcal{R}}$ by a tree labeled with the rewriting rules from some rewriting sequence $\mathbf{b} \leftarrow_{\mathcal{R}}^* \mathbf{t}$. As will be shown below, each such *rewriting tree* (Definition 3) defines an injective valuation $\nu : \mathbb{V}_1 \rightarrow [1, \kappa]^*$ of the bound variables from \mathbf{t} (Definition 4) that, in turn, induces a behavior $\mathbf{B}_{\nu}^{\mathbf{t}} \in \llbracket \mathbf{b} \rrbracket_{\mathcal{R}}$ (7). Since each term $\mathbf{t} \in \llbracket \mathbf{b} \rrbracket_{\mathcal{R}}$ can be represented by a rewriting tree (Proposition 1), it follows that each behavior $\mathcal{N} \in \llbracket \mathbf{b} \rrbracket_{\mathcal{R}}$ corresponds to a rewriting tree (up to a permutation of component identifiers). Our encoding uses rewriting trees \mathcal{T} as backbone parameters for the definition of the trap ($\Theta(\mathcal{N}(\mathcal{T}))$) and mutex ($\Omega(\mathcal{N}(\mathcal{T}))$) invariant, respectively, where $\mathcal{N}(\mathcal{T}) = \mathbf{B}_{\nu}^{\mathbf{t}}$ is the behavior induced by \mathcal{T} . In fact, any injective valuation of the variables is sufficient for safety checking, provided that the safety properties considered are defined by sets of error markings that are closed under permutations of component identifiers.

To simplify technicalities, we assume the existence of a rule $\mathbf{A}_{\mathbf{b}}(x_1, \dots, x_n) \leftarrow \mathbf{b}$ in \mathcal{R} , where $\text{fv}(\mathbf{b}) = \{x_1, \dots, x_n\}$ and $\mathbf{A}_{\mathbf{b}}$ is a predicate symbol of arity n not occurring elsewhere in \mathcal{R} . We also assume that the constant κ is greater than the number of predicate atoms that occur in any rule of \mathcal{R} .

Definition 3. *Given a rewriting system \mathcal{R} and a behavioral term \mathbf{b} , a rewriting tree for \mathbf{b} is a tree $\mathcal{T} : [1, \kappa]^* \rightarrow \mathcal{R}$, such that:*

1. $\mathcal{T}(\epsilon) = (\mathbf{A}_{\mathbf{b}}(x_1, \dots, x_n) \leftarrow \mathbf{b})$

and, for all nodes $w \in \text{nodes}(\mathcal{T})$, such that $\mathcal{T}(w) = (\mathbf{A}_w(x_1, \dots, x_{\#(\mathbf{A}_w)}) \leftarrow \mathbf{b}_w)$, the following hold:

2. for all $i \in [1, \#_{\text{pred}}(\mathbf{b}_w)]$, if $\text{pred}_i(\mathbf{b}_w) = \mathbf{A}_{w_i}(y_1, \dots, y_{\#(\mathbf{A}_{w_i})})$ then $w_i \in \text{nodes}(\mathcal{T})$ and $\mathcal{T}(w_i) = (\mathbf{A}_{w_i}(x_1, \dots, x_{\#(\mathbf{A}_{w_i})}) \leftarrow \mathbf{b}_{w_i})$, for some rule of the form $\mathbf{A}_{w_i}(x_1, \dots, x_{\#(\mathbf{A}_{w_i})}) \leftarrow \mathbf{b}_{w_i}$ from \mathcal{R} .
3. for all $i \geq \#_{\text{pred}}(\mathbf{b}_w)$, we have $w_i \notin \text{nodes}(\mathcal{T})$.

We denote by $\mathbb{T}_{\mathcal{R}}(\mathbf{b})$ the set of rewriting trees for \mathbf{b} in \mathcal{R} .

A rewriting tree \mathcal{T} induces a *characteristic term* $\mathfrak{C}_{[\mathcal{T}]}$, obtained by the application of the rewriting rules labeling the tree nodes in some order of traversal, and a *characteristic valuation* $\nu_{[\mathcal{T}]}$, that maps each variable in the term to the node where it is instantiated.

Definition 4. *Given a rewriting tree $\mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathbf{b})$, the characteristic term $\mathfrak{C}_{[\mathcal{T}]}$ and characteristic valuation $\nu_{[\mathcal{T}]}$ are defined inductively on the structure of \mathcal{T} :*

- if $\text{nodes}(\mathcal{T}) = \{\epsilon\}$, then $\mathfrak{C}_{[\mathcal{T}]} \stackrel{\text{def}}{=} \mathcal{B}(x)$ and $\nu_{[\mathcal{T}]}(x) \stackrel{\text{def}}{=} \epsilon$, for $\mathcal{T}(\epsilon) = (\mathbf{A}(x) \leftarrow \mathcal{B}(x))$,
- else, let $1, \dots, m$ be the children of the root of \mathcal{T} and let:

$$\mathfrak{C}_{[\mathcal{T}]} \stackrel{\text{def}}{=} \nu y_1 \dots \nu y_n \cdot \langle \Gamma \rangle (\mathfrak{C}_{[\mathcal{T}_{i1}]}[z_1^1/x_1, \dots, z_{\#(\mathbf{A}_1)}^1/x_{\#(\mathbf{A}_1)}], \dots, \mathfrak{C}_{[\mathcal{T}_{im}]}[z_1^m/x_1, \dots, z_{\#(\mathbf{A}_m)}^m/x_{\#(\mathbf{A}_m)}])$$

where $\nu_{[\mathcal{T}]}(z_j^i) \stackrel{\text{def}}{=} i \cdot \nu_{[\mathcal{T}_{i1}]}(x_j)$, for all $i \in [1, m]$ and $j \in [1, \#(\mathbf{A}_i)]$, such that

$$\mathcal{T}(\epsilon) = (\mathbf{A}(x_1, \dots, x_{\#(\mathbf{A})}) \leftarrow \nu y_1 \dots \nu y_n \cdot \langle \Gamma \rangle (\mathbf{A}_1(z_1^1, \dots, z_{\#(\mathbf{A}_1)}^1), \dots, \mathbf{A}_m(z_1^m, \dots, z_{\#(\mathbf{A}_m)}^m)))$$

A consequence of the specific form of rewriting rules (Definition 1) is that each (free or bound) variable y from $\mathfrak{C}_{[\mathcal{T}]}$ is mapped by $\nu_{[\mathcal{T}]}$ to a unique node $w \in \text{nodes}(\mathfrak{C}_{[\mathcal{T}]})$, such that $\mathfrak{C}_{[\mathcal{T}]}(w)$ contains an instance atom $\mathcal{B}(x)$, where x is substituted with y along the path from the node u that introduces y (take $u = \epsilon$ if $y \in \text{fv}(\mathfrak{C}_{[\mathcal{T}]})$) to w . For example, Fig. 3 shows the rewriting tree for the rewriting sequence from Example 3; we annotate on the side of the tree the bottom-up definition of the characteristic valuation associating the bound variables y_1 and y_2 with the nodes of the rewriting tree where they are instantiated.

Below we show that the set $[\mathbf{b}]_{\mathcal{R}}$ of predicateless terms obtained by complete rewriting is the same as the set of characteristic terms that correspond to some rewriting tree:

Proposition 1. *Given a behavioral term \mathbf{b} , we have $[\mathbf{b}]_{\mathcal{R}} = \{\mathfrak{C}_{[\mathcal{T}]} \mid \mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathbf{b})\}$.*

Its proof is an easy consequence of the confluence of the rewriting system, i.e. the order in which the rules are applied to a term does not change the resulting predicateless term.

4.1 Encoding Invariants and Error Configurations

We begin by building a $\text{WS}\kappa\text{S}$ formula that describes an infinite κ -ary tree whose finite prefix encodes a rewriting tree $\mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathbf{b})$. Let us assume that $\mathcal{R} = \{\mathbf{r}_1, \dots, \mathbf{r}_N\}$, such that $\mathbf{r}_1 = (\mathbf{A}_{\mathbf{b}}(x_1, \dots, x_n) \leftarrow \mathbf{b})$. We use a designated tuple of second order variables $\mathbf{U} = \langle U_1, \dots, U_N \rangle$, where each variable U_i is interpreted as the set of tree nodes labeled with the rule \mathbf{r}_i in \mathcal{T} . With this convention, the $\text{RTree}(\mathbf{U})$ formula (Fig. 4) defines a rewriting tree:

$$RTree(\mathbf{U}) \stackrel{\text{def}}{=} \forall x . \bigwedge_{1 \leq i < j \leq N} \left(\neg U_i(x) \vee \neg U_j(x) \right) \wedge U_1(x) \leftrightarrow x = \epsilon \wedge \quad (8)$$

$$\forall x . \bigwedge_{i:r_i \in \mathcal{R}} \bigwedge_{\ell=1}^{\kappa} U_i(x.\ell) \rightarrow \bigvee_{r_j \in \mathcal{R}} U_j(x) \wedge \quad (9)$$

$$\forall x . \bigwedge_{i:r_i = (A'(x_1, \dots, x_{\#(A')}) \leftarrow b')} \bigwedge_{j=1}^{\#_{\text{pred}}(b')} U_i(x) \rightarrow \bigvee_{\substack{j:\text{pred}_j(b') = A''(\xi_1, \dots, \xi_{\#(A'')}) \\ \ell:r_\ell = (A''(x_1, \dots, x_{\#(A'')}) \leftarrow b'')}} U_\ell(x.j) \wedge \quad (10)$$

$$\forall x . \bigwedge_{i:r_i = (A'(x_1, \dots, x_{\#(A')}) \leftarrow b')} \bigwedge_{j=\#_{\text{pred}}(b')+1}^{\kappa} U_i(x) \rightarrow \bigwedge_{\ell=1}^N \neg U_\ell(x.j) \quad (11)$$

Fig. 4. The Definition of Rewriting Trees

- line (8) states that the sets \mathbf{U} are pairwise disjoint and that U_1 is a singleton containing the root of the tree (condition 1 of Definition 3).
- line (9) states that the union of the sets \mathbf{U} is prefix-closed, i.e. the parent x of each node $x.\ell$ from some U_i belongs to some U_j , for $i, j \in [1, N]$.
- lines (10) and (11) encode the conditions 2 and 3 of Definition 3, respectively.

Clearly, for each model ν of $RTree(\mathbf{U})$, there is a unique rewriting tree, denoted $\mathcal{T}_\nu^U \in \mathbb{T}_{\mathcal{R}}(\mathbf{b})$, such that $\text{nodes}(\mathcal{T}_\nu^U) = \bigcup_{i=1}^N \nu(U_i)$ and $\mathcal{T}_\nu^U(w) = r_i$ iff $w \in \nu(U_i)$, for all $i \in [1, N]$ and $w \in \text{nodes}(\mathcal{T}_\nu^U)$.

As said, a rewriting tree $\mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathbf{b})$ defines a behavior (i.e. a marked PN) denoted by $\mathcal{N}(\mathcal{T}) \stackrel{\text{def}}{=} \mathbf{B}_\nu^t$ (6), where $t = \mathfrak{C}_{[\mathcal{T}]}$ is the characteristic term and $\nu = \nu_{[\mathcal{T}]}$ is the characteristic valuation of \mathcal{T} (Definition 4). An invariant of $\mathcal{N}(\mathcal{T})$ is a set of markings, i.e. a set of sets of marked places from different components.

Let $\{\mathcal{B}_i \stackrel{\text{def}}{=} \langle \mathcal{P}_i, \mathcal{S}_i, \mathcal{I}_i, \Delta_i \rangle\}_{i=1}^K$ be the set of component types that occur in the rules of \mathcal{R} and let $\mathbf{Z} = \langle Z_1, \dots, Z_K \rangle$ be a tuple of second-order variables, where Z_i is interpreted as the set of identifiers of the components of type \mathcal{B}_i . We encode the markings of $\mathcal{N}(\mathcal{T})$ by a WS κ S formula using a tuple of second-order variables $\mathbf{X} = \langle X_S \mid S \in \bigcup_{i=1}^K \mathcal{S}_i \rangle$, where each X_S is interpreted as the set of identifiers of the components currently in state $S(w)$ and define the set $\sigma_\nu^{\mathbf{x}} \stackrel{\text{def}}{=} \{S(w) \mid S \in \bigcup_{i=1}^K \mathcal{S}_i, w \in \nu(X_S)\}$. The following formula constrains the set represented by \mathbf{X} to be a marking of $\mathcal{N}(\mathcal{T}_\nu^U)$:

$$\begin{aligned} \text{mark}(\mathbf{X}, \mathbf{Z}) &\stackrel{\text{def}}{=} \forall x . \bigwedge_{S \neq S' \in \bigcup_{j=1}^K \mathcal{S}_j} (\neg X_S(x) \vee \neg X_{S'}(x)) \wedge \bigvee_{S \in \bigcup_{j=1}^K \mathcal{S}_j} X_S(x) \leftrightarrow \bigvee_{j=1}^K Z_j(x) \\ \text{inst}(\mathbf{Z}, \mathbf{U}) &\stackrel{\text{def}}{=} \forall x . \bigwedge_{i=1}^K Z_i(x) \leftrightarrow \bigvee_{j:r_j = (A'(y) \leftarrow \mathcal{B}_i(y))} U_j(x) \end{aligned}$$

Intuitively, $\text{mark}(\mathbf{X}, \mathbf{Z})$ states that no component can be in two different states (first conjunct) and each component is an instance of some component type (second conjunct). The formula $\text{inst}(\mathbf{Z}, \mathbf{U})$ above relates the instance indices to the nodes of the rewriting tree where the corresponding instance atoms occur,

assuming that the sets \mathbf{U} are constrained by $RTree(\mathbf{U})$. Then, for each model ν of $mark(\mathbf{X}, \mathbf{Z}) \wedge inst(\mathbf{Z}, \mathbf{U}) \wedge RTree(\mathbf{U})$, the set $\sigma_\nu^{\mathbf{X}}$ is a marking of $\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})$.

We proceed with the encoding of invariants and error states, by assuming the existence of a *flow formula*, that defines the pre- and post-sets of the transitions from a behavior $\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})$, formally described next (Sect. 4.2). In the following, the primed copy of the tuple \mathbf{X} is denoted as \mathbf{X}' .

Definition 5. $\Phi(\mathbf{X}, \mathbf{X}', \mathbf{U})$ is a flow formula for \mathbf{b} and \mathcal{R} if, for each model ν of $RTree(\mathbf{U})$, we have $\nu \models \Phi(\mathbf{X}, \mathbf{X}', \mathbf{U})$ if and only if $\bullet t = \sigma_\nu^{\mathbf{X}}$ and $t^\bullet = \sigma_\nu^{\mathbf{X}'}$, for some transition t of $\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})$.

Given a flow formula Φ , the parametric trap invariant $TrapInv^\Phi(\mathbf{X}, \mathbf{U})$ is defined by the formula below:

$$\begin{aligned} trap^\Phi(\mathbf{X}, \mathbf{U}) &\stackrel{\text{def}}{=} \forall \mathbf{Y}^1, \mathbf{Y}^2. \Phi(\mathbf{Y}^1, \mathbf{Y}^2, \mathbf{U}) \wedge inter(\mathbf{X}, \mathbf{Y}^1) \rightarrow inter(\mathbf{X}, \mathbf{Y}^2) \\ TrapInv^\Phi(\mathbf{X}, \mathbf{U}) &\stackrel{\text{def}}{=} \exists \mathbf{Z}. mark(\mathbf{X}, \mathbf{Z}) \wedge \forall \mathbf{Y}^1, \mathbf{Y}^2. init(\mathbf{Y}^1, \mathbf{Z}) \wedge inter(\mathbf{Y}^1, \mathbf{Y}^2) \wedge \\ &\quad trap^\Phi(\mathbf{Y}^2, \mathbf{U}) \rightarrow inter(\mathbf{X}, \mathbf{Y}^2) \end{aligned}$$

where the tuples $\mathbf{Y}^i \stackrel{\text{def}}{=} \langle Y_S^i \mid S \in \bigcup_{j=1}^K \mathcal{S}_j \rangle$, for $i = 1, 2$, are distinct copies of \mathbf{X} . The auxiliary formula $init(\mathbf{X}, \mathbf{Z}) \stackrel{\text{def}}{=} mark(\mathbf{X}, \mathbf{Z}) \wedge \bigwedge_{j=1}^K \forall x. Z_j(x) \leftrightarrow X_{\mathcal{I}_j}(x)$ states that \mathbf{X} represents the initial marking of the behavior, whereas $inter(\mathbf{X}, \mathbf{Y}) \stackrel{\text{def}}{=} \exists x. \bigvee_{j=1}^K \bigvee_{S \in \mathcal{S}_j} X_S(x) \wedge Y_S(x)$ means that the sets of places encoded by \mathbf{X} and \mathbf{Y} , respectively, have a non-empty intersection. Intuitively, the formula $trap^\Phi(\mathbf{X}, \mathbf{U})$ defines the traps (point 1 of Definition 2), whereas $TrapInv^\Phi(\mathbf{X}, \mathbf{U})$ defines the set of markings that intersect with the initial marking and with each trap of the behavior, i.e. the trap invariant (point A of Definition 2).

Mutexes and mutex invariants (points 2 and B of Definition 2) are defined by the formulæ:

$$\begin{aligned} mutex^\Phi(\mathbf{X}, \mathbf{U}) &\stackrel{\text{def}}{=} \forall \mathbf{Y}^1, \mathbf{Y}^2. \Phi(\mathbf{Y}^1, \mathbf{Y}^2, \mathbf{U}) \rightarrow \bigwedge \begin{array}{l} \neg inter(\mathbf{X}, \mathbf{Y}^1) \leftrightarrow \neg inter(\mathbf{X}, \mathbf{Y}^2) \\ single(\mathbf{X}, \mathbf{Y}^1) \leftrightarrow single(\mathbf{X}, \mathbf{Y}^2) \end{array} \\ MutexInv^\Phi(\mathbf{X}, \mathbf{U}) &\stackrel{\text{def}}{=} \exists \mathbf{Z}. mark(\mathbf{X}, \mathbf{Z}) \wedge \forall \mathbf{Y}^1, \mathbf{Y}^2. init(\mathbf{Y}^1, \mathbf{Z}) \wedge single(\mathbf{Y}^1, \mathbf{Y}^2) \wedge \\ &\quad mutex^\Phi(\mathbf{Y}^2, \mathbf{U}) \rightarrow single(\mathbf{X}, \mathbf{Y}^2) \end{aligned}$$

where $single(\mathbf{X}, \mathbf{Y}) \stackrel{\text{def}}{=} \exists_1 x. \bigvee_{j=1}^K \bigvee_{S \in \mathcal{S}_j} X_S(x) \wedge Y_S(x)$ states that the intersection of the sets of places defined by \mathbf{X} and \mathbf{Y} is a singleton³. The following lemma states the correctness of the encoding:

Lemma 1. *Given a flow formula $\Phi(\mathbf{X}, \mathbf{X}', \mathbf{U})$ and a model ν of $RTree(\mathbf{U}) \wedge inst(\mathbf{Z}, \mathbf{U})$, we have:*

1. $\Theta(\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})) = \{\sigma_\mu^{\mathbf{X}} \mid \mu \models TrapInv^\Phi(\mathbf{X}, \mathbf{U}), \mu(\mathbf{U} \cdot \mathbf{Z}) = \nu(\mathbf{U} \cdot \mathbf{Z})\}$,
2. $\Omega(\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})) = \{\sigma_\mu^{\mathbf{X}} \mid \mu \models MutexInv^\Phi(\mathbf{X}, \mathbf{U}), \mu(\mathbf{U} \cdot \mathbf{Z}) = \nu(\mathbf{U} \cdot \mathbf{Z})\}$.

³ $\exists_1 x. \phi(x)$ is a shorthand for $\exists x. \phi(x) \wedge \forall x \forall y. \phi(x) \wedge \phi(y) \rightarrow x = y$.

In our examples (Sect. 5) we consider two kinds of error sets, defined as:

$$\begin{aligned} \text{DeadLock}^\Phi(\mathbf{X}, \mathbf{U}) &\stackrel{\text{def}}{=} \forall \mathbf{Y}^1, \mathbf{Y}^2. \Phi(\mathbf{Y}^1, \mathbf{Y}^2, \mathbf{U}) \rightarrow \exists x. \bigvee_{j=1}^K \bigvee_{S \in \mathcal{S}_j} Y_S^1(x) \wedge \neg X_S(x) \\ \text{CriticalSection}^\Xi(\mathbf{X}, \mathbf{U}) &\stackrel{\text{def}}{=} \exists x \exists y. \bigvee_{S, S' \in \Xi} X_S(x) \wedge X_{S'}(y) \wedge \neg x = y \end{aligned}$$

Intuitively, $\text{DeadLock}^\Phi(\mathbf{X}, \mathbf{U})$ defined the deadlock markings, in which no transition of the behavior is enabled and $\text{CriticalSection}^\Xi(\mathbf{X}, \mathbf{U})$ states that no two components are at the same time in a state from a given set $\Xi \subseteq \bigcup_{i=1}^K \mathcal{S}_i$ of state types. It is worth mentioning that these sets of error markings are closed under permutations of component indices⁴, which makes them suitable for safety checking using our encoding of trap and mutex invariants (with variables mapped to the nodes of the rewriting tree where they occur instantiated, as in Fig. 3).

4.2 The Flow of a Behavioral Term

The previous definition of structural invariants relied on the existence of a flow formula $\Phi(\mathbf{X}, \mathbf{X}', \mathbf{U})$, stating that $\sigma_\nu^{\mathbf{X}}$ and $\sigma_\nu^{\mathbf{X}'}$ are the pre- and post-sets of some transition from the behavior (i.e. marked PN) $\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})$, whenever ν is a model of $RTree(\mathbf{U}) \wedge \Phi(\mathbf{X}, \mathbf{X}', \mathbf{U})$ (Definition 5). In this section, we describe the flow formula. In the following, we assume w.l.o.g. that the inductive rules in \mathcal{R} are of the form:

$$A(x_1, \dots, x_{\#(A)}) \leftarrow \nu y_1 \dots \nu y_m \cdot \left\langle \sum_{k=1}^m P_{k1}(x_{k1}) \cdot \dots \cdot P_{kr_k}(x_{kr_k}) \right\rangle (\mathbf{t}_1, \dots, \mathbf{t}_n)$$

if necessary, by applying the flattening relation (5) to each rule of \mathcal{R} . We denote by $\text{Inter}(\mathbf{r}) \stackrel{\text{def}}{=} \{\{P_{ki}(x_{ki}) \mid i \in [1, r_k]\} \mid k \in [1, m]\}$ the set of sets of port atoms $P_{ki}(x_{ki})$, corresponding to the interactions (i.e. the monomials from the architecture) from \mathbf{r} .

$$\Phi(\mathbf{X}, \mathbf{X}', \mathbf{U}) \stackrel{\text{def}}{=} \bigvee_{\ell=1}^N \bigvee_{\pi \in \text{Inter}(r_\ell)} \Psi_{\ell, \pi}(\mathbf{X}, \mathbf{X}', \mathbf{U}) \quad (12)$$

$$\Psi_{\ell, \{P_1(x_1), \dots, P_n(x_n)\}}(\mathbf{X}, \mathbf{X}', \mathbf{U}) \stackrel{\text{def}}{=} \exists y_0 \dots \exists y_n. U_\ell(y_0) \wedge \quad (13)$$

$$\bigwedge_{i=1}^n \left(\bigvee_{r'=(A'(y_i) \leftarrow \mathcal{B}(y_i))} \text{Path}_{r_\ell, r'}^{x_i, y_i}(y_0, y_i, \mathbf{U}) \right) \wedge \quad (14)$$

$$\forall x. \bigwedge_{S \in \bigcup_{j=1}^K \mathcal{S}_j} \left[\left(X_S(x) \leftrightarrow \bigvee_{\bullet P_k=S} x = y_k \right) \wedge \left(X'_S(x) \leftrightarrow \bigvee_{P_k \bullet=S} x = y_k \right) \right] \quad (15)$$

Fig. 5. Definition of the Flow Formula for the Rewriting System $\mathcal{R} = \{r_1, \dots, r_N\}$.

⁴ This is the case for every WS κ S formula consisting of equality and membership atoms, without successor functions.

The flow formula $\Phi(\mathbf{X}, \mathbf{X}', \mathbf{U})$ is given in Fig. 5. Essentially, the formula (12) is split into a disjunction of formulæ $\Psi_{\ell, \{P_1(x_1), \dots, P_n(x_n)\}}$ (13), one for each rule $r_\ell \in \mathcal{R}$ and each set of port atoms $\{P_1(x_1), \dots, P_n(x_n)\}$, denoting an interaction (monomial) from r_ℓ . To understand the formulæ (13), recall that each of the variables x_1, \dots, x_n is mapped to the (unique) node of the rewriting tree containing an instance atom $\mathcal{B}_i(x_i)$. In order to find this node, we must track the variable x_i from the node labeled by the rule r , to the node where this instance atom occurs. This is done by the $Path_{r_\ell, r'}^{x_i, y_i}(y_0, y_i, \mathbf{U})$ formulæ (14), that holds whenever $\mathcal{T}_\nu^\mathbf{U}$ is a rewriting tree, uniquely encoded by the interpretation ν of the \mathbf{U} variables, and y_0, y_i are mapped to the source and the destination of a path from a node $w \in \text{nodes}(\mathcal{T}_\nu^\mathbf{U})$, with label $\mathcal{T}_\nu^\mathbf{U}(w) = r_\ell$ to a node $w' \in \text{nodes}(\mathcal{T}_\nu^\mathbf{U})$, with label $\mathcal{T}_\nu^\mathbf{U}(w') = r'$, such that x_i and y_i are variables that occur in the bodies of r and r' , respectively, mapped to the same variable in the characteristic term of $\mathcal{T}_\nu^\mathbf{U}$ (Definition 4). We describe these paths by an automaton and define $Path_{r_\ell, r'}^{x_i, y_i}(y_0, y_i, \mathbf{U})$, by translating this automaton into a WS κ S formula.

But first, let us define paths in a tree formally. Given a κ -ary tree \mathcal{T} , a *path* is a finite sequence of nodes $\rho = n_1, \dots, n_\ell \in \text{nodes}(\mathcal{T})$, such that, for all $i \in [1, \ell - 1]$, n_{i+1} is either the parent ($n_i = n_{i+1}\alpha_i$) or a child ($n_{i+1} = n_i\alpha_i$) of n_i , for some $\alpha_i \in [1, \kappa]$. The path is determined by the source node and the sequence $(\alpha_1, d_1) \dots (\alpha_{\ell-1}, d_{\ell-1})$ of *directions* $(\alpha_i, d_i) \in [1, \kappa] \times \{\uparrow, \downarrow\}$, with the following meaning: $d_i = \uparrow$ if $n_{i+1}\alpha_i = n_i$ and $d_i = \downarrow$ if $n_{i+1} = n_i\alpha_i$. Given two distinct nodes $w_1, w_2 \in \text{nodes}(\mathcal{T})$, there is a unique minimal path from w_1 to w_2 , labeled by a sequence denoted as $\rho(w_1, w_2)$. This path climbs from w_1 to the greatest common prefix w of w_1 and w_2 , before descending from w to w_2 .

A *path automaton* is a tuple $A = (Q, I, F, \delta)$, where Q is a set of states, $I, F \subseteq Q$ are the sets of initial and final states, respectively, and $\delta \subseteq Q \times [1, \kappa] \times \{\uparrow, \downarrow\} \times Q$ is a set of transitions $q \xrightarrow{(\alpha, d)} q'$, with $\alpha \in [1, \kappa]$ being a direction and $d \in \{\uparrow, \downarrow\}$ indicating whether the automaton moves up or down in the tree. A run of A over $\omega = (\alpha_1, d_1) \dots (\alpha_{n-1}, d_{n-1})$ is a sequence of states $q_1, \dots, q_n \in Q$ such that $q_1 \in I$ and $q_i \xrightarrow{(\alpha_i, d_i)} q_{i+1} \in \delta$, for all $i \in [1, n - 1]$. The run is accepting if and only if $q_n \in F$. The *language* $\mathcal{L}(A)$ of A is the set of sequences over which A has an accepting run.

A path automaton $A = (Q, I, F, \delta)$ corresponds (Lemma 2) to the formula in Fig. 6, that can be effectively built from the description of A . Here $Q = \{q_1, \dots, q_L\}$ is the set of states of A and $\mathbf{Y} = \langle Y_1, \dots, Y_L \rangle$ are second order variables interpreted as the sets of tree nodes labeled by the automaton with q_1, \dots, q_L , respectively. Intuitively, the first three conjuncts of the above formula (16) encode the facts that \mathbf{Y} are disjoint (no tree node is labeled by more than one state during the run) and that the run starts in an initial state with node x and ends in a final state with node y . The fourth conjunct (17) states that, for every non-final node on the path, if the automaton visits that node by state q_i , then either the node has a (α, \downarrow) -child or a (α, \uparrow) -parent visited by state q_j , where $q_i \xrightarrow{(\alpha, \downarrow)} q_j$ and $q_i \xrightarrow{(\alpha, \uparrow)} q_j$ are transitions of the automaton. The fifth conjunct (18) is the reversed flow condition on the path, needed to ensure that

$$\begin{aligned}
 \Delta_A(x, y, \mathbf{Y}) &\stackrel{\text{def}}{=} \bigwedge_{1 \leq i \neq j \leq L} \forall z. (\neg Y_i(z) \vee \neg Y_j(z)) \wedge \bigvee_{q_i \in I} Y_i(x) \wedge \bigvee_{q_j \in F} Y_j(y) \wedge \quad (16) \\
 &\bigwedge_{i=1}^L \forall z. z \neq y \wedge Y_i(z) \rightarrow \\
 &\quad \bigvee_{j: q_i \xrightarrow{(\alpha, \downarrow)} q_j} Y_j(z.\alpha) \vee \bigvee_{j: q_i \xrightarrow{(\alpha, \uparrow)} q_j} \exists z'. z'.\alpha = z \wedge Y_j(z') \wedge \quad (17) \\
 &\bigwedge_{j=1}^L \forall z. z \neq x \wedge Y_j(z) \rightarrow \\
 &\quad \bigvee_{i: q_i \xrightarrow{(\alpha, \downarrow)} q_j} \exists z'. z'.\alpha = z \wedge Y_i(z') \vee \bigvee_{i: q_i \xrightarrow{(\alpha, \uparrow)} q_j} Y_i(z.\alpha) \quad (18)
 \end{aligned}$$

Fig. 6. Definition of the Path Automaton formula $\Delta_A(x, y, \mathbf{Y})$

the sets \mathbf{Y} do not contain useless nodes, being thus symmetric to the fourth. The following result stems from the classical automata-logic connection⁵ [14, §2.10]:

Lemma 2. *Given a tree \mathcal{T} with $\text{nodes}(\mathcal{T}) \subseteq [1, \kappa]^*$ and a sequence $\omega \in ([1, \kappa] \times \{\uparrow, \downarrow\})^*$ from $w_1 \in \text{nodes}(\mathcal{T})$ to $w_2 \in \text{nodes}(\mathcal{T})$, for each valuation ν such that $\nu(x) = w_1$ and $\nu(y) = w_2$, we have $\omega \in \mathcal{L}(A) \iff \nu \models \exists \mathbf{Y}. \Delta_A(x, y, \mathbf{Y})$.*

Our purpose is to define path automata that recognize the paths between the node where a bound variable is introduced and the node where the variable is instantiated, in a given rewriting tree. This automaton is directly inferred from the syntax of the rules in \mathcal{R} . For each pair of rules $r_1, r_2 \in \mathcal{R}$ and variables $z_1, z_2 \in \mathbb{V}_1$ that occur in the bodies of r_1 and r_2 , respectively, we define the path automaton $A_{r_1, r_2}^{z_1, z_2} \stackrel{\text{def}}{=} (Q, I_{r_1}^{z_1}, F_{r_2}^{z_2}, \delta)$:

- We associate a state $q_{r,z}^d$ to each rule $r = (A(x_1, \dots, x_{\#A}) \leftarrow \bar{b})$, each variable z occurring (free or bound) in \bar{b} and each direction $d \in \{\uparrow, \downarrow\}$. The intuition is that the automaton visits the state $q_{r,z}^d$ while going up or down, as indicated by the direction d , currently tracking variable z in rule r .

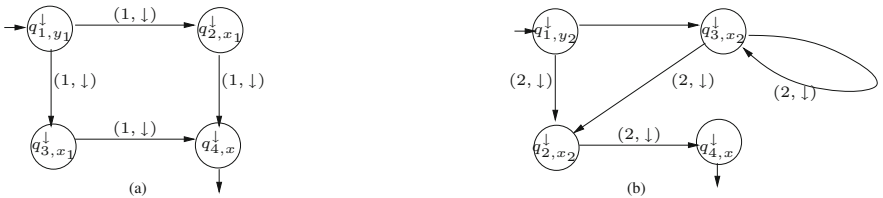


Fig. 7. Path Automata Recognizing the Instantiation Paths from Example 1

⁵ A similar conversion of tree walking automata to MSO has been described in [12].

- The sets of initial and final states are $I_{r_1}^{z_1} \stackrel{\text{def}}{=} \{q_{r_1, z_1}^d \mid d = \uparrow, \downarrow\}$ and $F_{r_2}^{z_2} \stackrel{\text{def}}{=} \{q_{r_2, z_2}^\downarrow\}$. In other words, the automaton starts to track z_1 in r_1 , moving either up or down and it ends tracking z_2 in r_2 , while moving down.
- The transitions are $q_{r_1, y_j}^\downarrow \xrightarrow{(\alpha, \downarrow)} q_{r_2, x_j}^\downarrow$, $q_{r_2, x_j}^\uparrow \xrightarrow{(\alpha, \uparrow)} q_{r_1, y_j}^\uparrow$ and $q_{r_2, x_j}^\uparrow \xrightarrow{(\alpha, \uparrow)} q_{r_1, y_j}^\downarrow$, for any two distinct rules $r_i = (A_j(x_1, \dots, x_{\#(A)}) \leftarrow b_i)$, $i = 1, 2$, all $\alpha \in [1, \#_{\text{pred}}(b_1)]$, such that $\text{pred}_\alpha(b_1) = A_2(y_1, \dots, y_{\#(A_2)})$ and all $j \in [1, \#(A_2)]$. Intuitively, if r_1 labels the parent of the node labeled by r_2 in the rewriting tree, the automaton can move either: (i) down from tracking y_j in r_1 to tracking x_j in r_2 , (ii) up from tracking x_j in r_2 to tracking y_j in r_1 , or (iii) change direction from moving up tracking x_j in r_2 to moving down tracking y_j in r_1 . In particular, the last case might be needed to accept a path that only goes up in the tree.

Note that a run of a path automaton $A_{r_1, r_2}^{z_1, z_2}$ may have at most one change of direction, by a rule of the form $q_{r_2, x_j}^\uparrow \xrightarrow{(\alpha, \uparrow)} q_{r_1, y_j}^\downarrow$.

Example 4. The paths that track the instantiations of the variables y_1 and y_2 in a rewriting tree for the term $\text{Ring}()$ are depicted in dashed lines in Fig. 3. The path automata that recognize these paths are given in Fig. 7a (y_1) and Fig. 7b (y_2). The initial states are q_{r_1, y_1}^\downarrow and q_{r_1, y_2}^\downarrow , respectively, and the final state is $q_{r_2, x}^\downarrow$ in both cases, where the labels (1–4) of the rewriting rules are the ones from Example 1. ■

The lemma below shows that these automata recognize exactly the labels of the minimal paths between two nodes:

Lemma 3. *Let $\mathcal{T} \in \mathbb{T}_{\mathcal{R}}(\mathbf{b})$ be a rewriting tree and $w_i \in \text{nodes}(\mathcal{T})$ be nodes labeled with the rules $\mathcal{T}(w_i) = (A_i(x_{i,1}, \dots, x_{i, \#(A_i)}) \leftarrow b_i) = r_i$, for $i = 1, 2$. Then, for all $k_i \in [1, \#(A_i)]$, $i = 1, 2$, the following are equivalent:*

1. x_{1, k_1} and x_{2, k_2} are substituted by the same variable during the construction of $\mathfrak{C}_{[\mathcal{T}]}$ (Definition 4),
2. $\rho(w_1, w_2) \in \mathcal{L}(A_{r_1, r_2}^{x_{1, k_1}, x_{2, k_2}})$.

The path automata $A_{r_1, r_2}^{x_{1, k_1}, x_{2, k_2}}$ are used to define the $\text{Path}_{r_1, r_2}^{z_1, z_2}$ formulæ:

$$\begin{aligned} \text{Path}_{r_1, r_2}^{z_1, z_2}(x, y, \mathbf{U}) &\stackrel{\text{def}}{=} \exists \mathbf{Y} . \Delta_{A_{r_1, r_2}^{z_1, z_2}}(x, y, \mathbf{Y}) \wedge \mathcal{T}(\mathbf{Y}, \mathbf{U}) \\ \mathcal{T}(\mathbf{Y}, \mathbf{U}) &\stackrel{\text{def}}{=} \bigwedge_{d=\uparrow, \downarrow} \bigwedge_{i: r_i = (A'(x_1, \dots, x_{\#(A')}) \leftarrow b')} \bigwedge_{z \in \text{fv}(b')} \forall x . Y_{r, z}^d(x) \rightarrow U_i(x) \end{aligned}$$

The formula $\mathcal{T}(\mathbf{Y}, \mathbf{U})$ above states that all nodes labeled with a state $q_{r, z}^d$ during the run must be also labeled with r in the rewriting tree given as input to the path automaton. The lemma below proves that $\Phi(\mathbf{X}, \mathbf{X}', \mathbf{U})$ (12) is indeed a flow formula (Definition 5):

Lemma 4. *For each model ν of $\text{RTree}(\mathbf{U})$, we have $\nu \models \Phi(\mathbf{X}, \mathbf{X}', \mathbf{U})$ if and only if $\sigma_\nu^{\mathbf{x}} = \bullet t$ and $\sigma_\nu^{\mathbf{x}'} = t \bullet$ for some transition t of $\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})$.*

Together with Lemma 1, this ensures that the trap and mutex invariant of the parametric system described by \mathbf{b} and \mathcal{R} are defined by the $TrapInv^\Phi(\mathbf{X}, \mathbf{U})$ and $MutexInv^\Phi(\mathbf{X}, \mathbf{U})$ formulæ, respectively. Hence a sufficient condition that proves a safety property of the parametric system described by \mathbf{b} and \mathcal{R} is the unsatisfiability of a $WS\kappa S$ formula, obtained from the syntax of \mathbf{b} and \mathcal{R} :

Theorem 1. *Let \mathbf{b} be closed behavioral term, \mathcal{R} be a rewriting system and $\mathcal{E}(\mathbf{X}, \mathbf{U})$ be a $WS\kappa S$ formula. The behavior $\mathcal{N}(\mathcal{T}_\nu^{\mathbf{U}})$ is safe w.r.t. the set $\{\sigma_\mu^{\mathbf{X}} \mid \mu \models \mathcal{E}(\mathbf{X}, \mathbf{U}), \mu(\mathbf{U}) = \nu(\mathbf{U})\}$, for any valuation ν , if the formula $RTree(\mathbf{U}) \wedge TrapInv^\Phi(\mathbf{X}, \mathbf{U}) \wedge MutexInv^\Phi(\mathbf{X}, \mathbf{U}) \wedge \mathcal{E}(\mathbf{X}, \mathbf{U})$ is unsatisfiable.*

In particular, we have experimented with error sets defined by the $DeadLock^\Phi(\mathbf{X}, \mathbf{U})$ and $CriticalSection^\Xi(\mathbf{X}, \mathbf{U})$ formulæ, for some critical section given by $\Xi \in \bigcup_{i=1}^K \mathcal{S}_i$.

5 Experimental Evaluation

We implemented the structural invariant synthesis in a prototype tool⁶. Table 1 shows the results of checking deadlock freedom in all test cases and absence of critical section violations, for those test cases where a critical section was defined (otherwise marked n/a). The 2nd column gives the number of states in the system, in the form $n_1 \times \dots \times n_K$, where n_i is the number of states in the i -th component type and K is the number of component types. The number of rewriting rules and interactions in the specification are given in the 3rd and 4th columns, respectively. The 5th and 7th columns report the results of the satisfiability check (\checkmark means that the formula is unsatisfiable and \times means that a counterexample has been found, in which case safety could not be proved using our method) for deadlock freedom and absence of critical section violations, using the MONA v1.4-18 tool [11]. The 6th and 8th columns show the total running times (in seconds) on an iMac 3,4 GHz with 32 GB of RAM, respectively (∞ means that MONA has run out of memory). The 9th column gives the branching degree $\kappa \in \{1, 2\}$ of trees in the $WS\kappa S$ logic. Note that star and token ring systems require $\kappa = 1$, whereas the tree-structured systems require $\kappa = 2$.

The test cases we consider are grouped according to the architectural pattern. *Token rings* (Fig. 1a) consist of instances of the same component type, such that the *out* port of a component is connected to the *in* port of the next component in the ring. *Dining philosophers* are special cases of token rings, consisting of alternating *philosopher* and *fork* instances. *Stars* consists of a single controller (master) sending requests and receiving replies from one or more slaves connected to it. Concerning *trees*, the *tree-dfs* example models a binary tree architecture traversed by a token in depth-first order, while *tree-back-root* and *tree-linked-leaves(-generic)* go beyond trees, modeling hierarchical systems with parent-children communication on top of which the nodes communicate with the root and the leaves are linked in a token-ring, respectively. These examples could

⁶ <https://github.com/radiusif/rtab>.

not have been described using first order logic, as in [4]. The verification problems considered could be solved in less than 1 s, with the exception of the critical section violations for the *tree-linked-leaves(-generic)* examples, that require mutex, in addition to trap invariants. In particular, in the examples marked with *-generic*, the initial state of the components is arbitrary. Consequently, all these examples violate the critical section initially.

Table 1. Experimental Results

Example	#states	#rules	#interaction types	deadlock freedom	time (secs)	critical (secs)	time (secs)	κ
token-ring	2×2	3	3	✓	0.66	✓	0.63	1
token-ring-generic	2×2	5	4	✓	0.75	×	0.72	1
sync-philos	2×2	3	6	✓	0.69	✓	0.67	1
alt-philos-sym	3×2	3	9	×	0.75	✓	0.77	1
alt-philos-asym	3×2	3	9	✓	0.84	✓	0.78	1
alt-philos-generic	3×2	4	12	✓	0.91	✓	0.87	1
star	2×2	3	4	✓	0.58	n/a	–	1
star-ring	$2 \times 3 \times 3$	3	9	✓	0.75	✓	0.76	1
star-ring-generic	$2 \times 3 \times 3$	5	12	✓	0.84	×	0.88	1
tree-dfs	$2 \times 6 \times 2$	4	6	✓	0.70	n/a	–	2
tree-back-root	2×2	3	5	✓	0.60	n/a	–	2
tree-linked-leaves	$2 \times 2 \times 4 \times 3$	4	10	✓	1.05	✓	1.21	2
tree-linked-leaves-generic	$2 \times 2 \times 4 \times 3$	7	16	✓	1.31	×	1.73	2

6 Related Work

Traditionally, verification of unbounded networks of parallel processes considers known architectural patterns, typically cliques or rings [6, 10]. Because the price for decidability is drastic restrictions on the shape of architectures [3], more recent works propose practical semi-algorithms, e.g. *regular model checking* [1, 13] or *automata learning* [7]. Here the architectural pattern is implicitly determined by the class of language recognizers: word automata encode pipelines or rings, whereas tree automata describe trees. A first attempt at specifying architectures by logic is the *interaction logic* of Konnov et al. [15], which is a combination of Presburger arithmetic with monadic uninterpreted function symbols, that can describe cliques, stars and rings. More structured architectures (pipelines and trees) can be described using a second-order extension [17]. As such, these interaction logics are undecidable and have no support for automated verification. Recently, interaction logics that support the verification of safety properties by structural invariant synthesis have been developed. These logics use fragments of first order logic with interpreted function symbols that implicitly determine the shape of the architecture [4, 5].

7 Conclusions and Future Work

We present a formal language for the specification of distributed systems parameterized by the number of replicated components and by the shape of the coordinating architecture. The language uses inductive definitions to describe systems of unbounded size. We propose a verification method for safety properties based on the synthesis of structural invariants able to prove deadlock freedom for a number of non-trivial models.






References

1. Abdulla, P.A., Delzanno, G., Henda, N.B., Rezine, A.: Regular model checking without transducers (on efficient verification of parameterized systems). In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_56
2. Barrett, C.W., Shikhanian, I., Tinelli, C.: An abstract decision procedure for a theory of inductive data types. *J. Satisf. Boolean Model. Comput.* **3**(1–2), 21–46 (2007)
3. Bloem, R., et al.: Decidability of Parameterized Verification. *Synthesis Lectures on Distributed Computing Theory*. Morgan & Claypool Publishers, San Rafael (2015)
4. Bozga, M., Esparza, J., Iosif, R., Sifakis, J., Welzel, C.: Structural invariants for the verification of systems with parameterized architectures. In: TACAS 2020. LNCS, vol. 12078, pp. 228–246. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_13
5. Bozga, M., Iosif, R., Sifakis, J.: Checking deadlock-freedom of parametric component-based systems. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11428, pp. 3–20. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_1
6. Browne, M., Clarke, E., Grumberg, O.: Reasoning about networks with many identical finite state processes. *Inf. Comput.* **81**(1), 13–31 (1989)
7. Chen, Y., Hong, C., Lin, A.W., Rümmer, P.: Learning to prove safety over parameterised concurrent systems. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, pp. 76–83. IEEE (2017)
8. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 269–282. ACM Press, New York (1979)
9. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: Cytron, R.K., Lee, P. (eds.) Conference Record of POPL 1995: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 85–94. ACM Press (1995)
10. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. *J. ACM* **39**(3), 675–735 (1992)
11. Henriksen, J.G., et al.: Mona: monadic second-order logic in practice. In: Brinksma, E., Cleaveland, W.R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 89–110. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60630-0_5
12. Iosif, R., Rogalewicz, A., Simacek, J.: The tree width of separation logic with recursive definitions. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 21–38. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_2

13. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. *Theoret. Comput. Sci.* **256**(1), 93–112 (2001)
14. Khoussainov, B., Nerode, A.: *Automata Theory and Its Applications*. Springer, New York (2001). <https://doi.org/10.1007/978-1-4612-0171-7>
15. Konnov, I.V., Kotek, T., Wang, Q., Veith, H., Bliudze, S., Sifakis, J.: Parameterized systems in BIP: design and model checking. In: Desharnais, J., Jagadeesan, R. (eds.) *27th International Conference on Concurrency Theory, CONCUR 2016*. LIPIcs, vol. 59, pp. 30:1–30:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
16. Kramer, J., Magee, J.: Analysing dynamic change in distributed software architectures. *IEE Proc. Softw.* **145**(5), 146–154 (1998)
17. Mavridou, A., Baranov, E., Bliudze, S., Sifakis, J.: Configuration logics: modeling architecture styles. *J. Log. Algebr. Meth. Program.* **86**(1), 2–29 (2017)
18. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pp. 55–74. IEEE Computer Society (2002)
19. Thatcher, J., Wright, J.: Generalized finite automata theory with an application to a decision problem of second-order logic. *Math. Syst. Theory* **2**, 57–81 (2005)



A Linear Parallel Algorithm to Compute Bisimulation and Relational Coarsest Partitions

Jan Martens¹ , Jan Friso Groote¹ , Lars van den Haak¹ ,
Pieter Hijma^{1,2} , and Anton Wijs¹ 

¹ Eindhoven University of Technology, Eindhoven, The Netherlands

{j.j.m.martens, j.f.groote, l.b.v.d.haak, a.j.wijs}@tue.nl

² Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
pieter@cs.vu.nl

Abstract. The most efficient way to calculate strong bisimilarity is by finding the relational coarsest partition of a transition system. We provide the first linear-time algorithm to calculate strong bisimulation using parallel random access machines (PRAMs). More precisely, with n states, m transitions and $|Act| \leq m$ action labels, we provide an algorithm for $\max(n, m)$ processors that calculates strong bisimulation in time $\mathcal{O}(n + |Act|)$ and space $\mathcal{O}(n + m)$. The best-known PRAM algorithm has time complexity $\mathcal{O}(n \log n)$ on a smaller number of processors making it less suitable for massive parallel devices such as GPUs. An implementation on a GPU shows that the linear time-bound is achievable on contemporary hardware.

1 Introduction

The notion of *bisimilarity* for Kripke structures and Labelled Transition Systems (LTSs) is commonly used to define behavioural equivalence. Deciding this behavioural equivalence is important in the field of modelling and verifying concurrent and multi-component systems [4, 15]. Kanellakis and Smolka proposed a partition refinement algorithm for obtaining the bisimilarity relation for Kripke structures [11]. The proposed algorithm has a run time complexity of $\mathcal{O}(nm)$ where n is the number of states and m is the number of transitions of the input. Later, a more sophisticated refinement algorithm running in $\mathcal{O}(m \log n)$ steps was proposed by Paige and Tarjan [16].

In recent years the increase in the speed of sequential chip designs has stagnated due to a multitude of factors such as energy consumption and heat generation. In contrast, parallel devices such as graphics processing units (GPUs) keep increasing rapidly in computational power. In order to profit from the acceleration of these devices, we require algorithms with massive parallelism. The article “There’s plenty of room at the Top: What will drive computer performance after

This work is carried out in the context of the NWO AVVA project 612.001751 and the NWO TTW ChEOPS project 17249.

© The Author(s) 2021

G. Salaün and A. Wijs (Eds.): FACS 2021, LNCS 13077, pp. 115–133, 2021.

https://doi.org/10.1007/978-3-030-90636-8_7

Moore’s law” by Leieron et al. [13] indicates that the advance in computational performance will come from software and algorithms that can employ hardware structures with a massive number of simple, parallel processors, such as GPUs. In this paper, we propose such an algorithm to decide bisimilarity.

Deciding bisimilarity is P -complete [1], which suggests that bisimilarity is an inherently sequential problem. This fact has not withheld the community from searching for efficient parallel algorithms for deciding bisimilarity of Kripke structures. In particular, Lee and Rajasekaran [12, 17] proposed a parallel algorithm based on the Paige Tarjan algorithm that works in $\mathcal{O}(n \log n)$ time complexity using $\frac{m}{\log n} \log \log n$ Concurrent, Read Concurrent Write (CRCW) processors, and one using only $\frac{m}{n} \log n$ Concurrent Read Exclusive Write (CREW) processors. Jeong et al. [10] presented a linear time parallel algorithm, but it is probabilistic in the sense that it has a non-zero chance to output the wrong result. Furthermore, Wijs [22] presented a GPU implementation of an algorithm to solve the strong and branching bisimulation partition refinement problems. In a distributed setting, Blom and Orzan studied algorithms for refinement [2]. Those algorithms use message passing as a way of communication between different workers in a network and rely on a small number of processors. Therefore, they are very different in nature than our algorithm. Those algorithms were extended and optimized for branching bisimulation [3].

In this work, we improve on the best known theoretical bound for PRAM algorithms using a higher degree of parallelism. The proposed algorithm improves the run time complexity to $\mathcal{O}(n)$ on $\max(n, m)$ processors and is based on the sequential algorithm of Kanellakis and Smolka [11]. The larger number of processors used in this algorithm favours the increasingly parallel design of contemporary and future hardware. In addition, the algorithm is *optimal* w.r.t. the sequential Kanellakis-Smolka algorithm, meaning that overall, it does not perform more work than its sequential counterpart.

We first present our algorithm on Kripke structures where transitions are unlabelled. However, as labelled transition systems (LTSs) are commonly used, and labels are not straightforward to incorporate in an efficient way (cf. for instance [21]), we discuss how our algorithm can be extended to take action labels into account. This leads to an algorithm with a run time complexity of $\mathcal{O}(n + |Act|)$, with Act the set of action labels.

Our algorithm has been designed for and can be analyzed with the CRCW PRAM model, following notations from [20]. This model is an extension of the normal RAM model, allowing multiple processors to work with shared memory. In the CRCW PRAM model, parallel algorithms can be described in a straightforward and elegant way. In reality, no device exists that completely adheres to this PRAM model, but with recent advancements, hardware gets better and better at approximating the model since the number of parallel threads keeps growing. We demonstrate this by translating the PRAM algorithm to GPU code. We straightforwardly implemented our algorithm in CUDA and experimented with an NVIDIA Titan RTX, showing that our algorithm performs mostly in line with what our PRAM algorithm predicts.

The paper is structured as follows: In Sect. 2, we recall the necessary preliminaries on the CRCW PRAM model and state the partition refinement problems this paper focuses on. In Sect. 3, we propose a parallel algorithm to compute bisimulation for Kripke structures, which is also called the Relational Coarsest Partition Problem (RCPP). In this section, we also prove the correctness of the algorithm and provide a complexity analysis. In Sect. 4, we discuss the details for an implementation with multiple action labels, thereby supporting LTSs, which forms the Bisimulation Coarsest Refinement Problem (BCRP). In Sect. 5 we discuss the results of the implementation and in Sect. 6 we draw conclusions.

2 Preliminaries

2.1 The PRAM Model

The *Parallel Random Access Machine* (PRAM) is a natural extension of the normal Random Access Machine (RAM), where an arbitrary number of parallel processors can access the memory. Following the definitions of [20] we use a version of PRAM that is able to Concurrently Read and Concurrently Write (CRCW PRAM). It differs from the model introduced in [6] in which the PRAM model was only allowed to concurrently read from the same memory address, but concurrent writes (to the same address) could not happen.

A CRCW PRAM consists of a sequence of numbered processors P_0, P_1, \dots . These processors have all the natural instructions of a normal RAM such as addition, subtraction, and conditional branching based on the equality and less-than operators. There is an infinite amount of common memory the processors have access to. The processors have instructions to read from and write to the common memory. In addition, a processor P_i has an instruction to obtain its unique index i . A PRAM also has a function $\mathcal{P} : \mathbb{N} \rightarrow \mathbb{N}$ which defines a bound on the number of processors given the size of the input.

All the processors have the same program and run synchronized in a single instruction, multiple data (SIMD) fashion. In other words, all processors execute the program in lock-step. Parallelism is achieved by distributing the data elements over the processors and having the processors apply the program instructions on ‘their’ data elements.

Initially, given input consisting of n data elements, the CRCW PRAM assumes that the input is stored in the first n registers of the common memory, and starts the first $\mathcal{P}(n)$ processors $P_0, P_1, \dots, P_{\mathcal{P}(n)-1}$.

Whenever a concurrent write happens to the same memory cell, we assume that one arbitrary write will succeed. This is called the *arbitrary* CRCW PRAM.

A parallel program for a PRAM is called *optimal* w.r.t. a sequential algorithm if the total work done by the program does not exceed the work done by the sequential algorithm. More precisely, if T is the parallel run time and P the number of processors used, then the algorithm is optimal w.r.t. a sequential algorithm running in S steps if $P \cdot T \in \mathcal{O}(S)$.

2.2 Strong Bisimulation

To formalise concurrent system behaviour, we use LTSs.

Definition 1 (Labeled Transition System). *A Labeled Transition System (LTS) is a three-tuple $A = (S, Act, \rightarrow)$ where S is a finite set of states, Act a finite set of action labels, and $\rightarrow \subseteq S \times Act \times S$ the transition relation.*

Let $A = (S, Act, \rightarrow)$ be an LTS. Then, for any two states $s, t \in S$ and $a \in Act$, we write $s \xrightarrow{a} t$ iff $(s, a, t) \in \rightarrow$.

Kripke structures differ from LTSs in the fact that the states are labelled as opposed to the transitions. In the current paper, for convenience, instead of using Kripke structures where appropriate, we reason about LTSs with a single action label, i.e., $|Act| = 1$. Computing the coarsest partition of such an LTS can be done in the same way as for Kripke structures, apart from the fact that in the latter case, a different initial partition is computed that is based on the state labels (see, for instance, [8,9]).

Definition 2 (Strong bisimulation). *On an LTS $A = (S, Act, \rightarrow)$ a relation $R \subseteq S \times S$ is called a strong bisimulation relation if and only if it is symmetric and for all $s, t \in S$ with sRt and for all $a \in Act$ with $s \xrightarrow{a} s'$, we have:*

$$\exists t' \in S. t \xrightarrow{a} t' \wedge s'Rt'$$

Whenever we refer to bisimulation we mean strong bisimulation. Two states $s, t \in S$ in an LTS A are called *bisimilar*, denoted by $s \simeq t$, iff there is some bisimulation relation R for A that relates s and t .

A *partition* π of a finite set of states S is a set of subsets that are pairwise disjoint and whose union is equal to S , i.e., $\bigcup_{B \in \pi} B = S$. Every element $B \in \pi$ of this partition π is called a *block*.

We call partition π' a *refinement* of π iff for every block $B' \in \pi'$ there is a block $B \in \pi$ such that $B' \subseteq B$. We say a partition π of a finite set S induces the relation $R_\pi = \{(s, t) \mid \exists B \in \pi. s \in B \wedge t \in B\}$. This is an equivalence relation of which the blocks of π are the equivalence classes.

Given an LTS $A = (S, Act, \rightarrow)$ and two states $s, t \in S$ we say that s *reaches* t with action $a \in Act$ iff $s \xrightarrow{a} t$. A state s *reaches* a set $U \subseteq S$ with an action a iff there is a state $t \in U$ such that s reaches t with action a . A set of states $V \subseteq S$ is called *stable* under a set of states $U \subseteq S$ iff for all actions a either all states in V reach U with a , or no state in V reaches U with a . A partition π is stable under a set of states U iff each block $B \in \pi$ is stable under U . The partition π is called *stable* iff it is stable under all its own blocks $B \in \pi$.

Fact 1. [16] *Stability is inherited under refinement, i.e. given a partition π of S and a refinement π' of π , then if π is stable under $U \subseteq S$, then π' is also stable under U .*

The main problem we focus on in this work is called the bisimulation refinement problem (**BCRP**). It is defined as follows:

Input: An LTS $M = (S, Act, \rightarrow)$.

Output: The partition π of S which is the coarsest partition, i.e., has the smallest number of blocks, that forms a bisimulation relation.

In a Kripke structure, the transition relation forms a single binary relation, since the transitions are unlabelled. This is also the case when an LTS has a single action label. In that case, the problem is called the Relational Coarsest Partition Problem (**RCPP**) [11, 12, 16]. This problem is defined as follows:

Input: A set S , a binary relation $\rightarrow: S \times S$ and an initial partition π_0

Output: The partition π which is the coarsest refinement of π_0 and which is a bisimulation relation.

It is known that BCRP is not significantly harder than RCPP as there are intuitive translations from LTSs to Kripke structures [5, Dfn. 4.1]. However, some non-trivial modifications can speed-up the algorithm for some cases, hence we discuss both problems separately. In Sect. 3, we discuss the basic parallel algorithm for RCPP, and in Sect. 4, we discuss the modifications required to efficiently solve the BCRP problem for LTSs with multiple action labels.

3 Relational Coarsest Partition Problem

In this section, we discuss a sequential algorithm based on the one of Kanellakis and Smolka [11] for RCPP (Sect. 3.1). This is the basic algorithm that we adapt to the parallel PRAM algorithm (Sect. 3.2). The algorithm starts with an input partition π_0 and refines all blocks until a stable partition is reached. This stable partition will be the coarsest refinement that defines a bisimulation relation.

3.1 The Sequential Algorithm

The sequential algorithm, Algorithm 1, works as follows. Given are a set S , a transition relation $\rightarrow \subseteq S \times S$, and an initial partition π_0 of S . Initially, we mark the partition as not necessarily stable under all blocks by putting these blocks in a set *Unstable*. In any iteration of the algorithm, if a block B of the current partition is not in *Unstable*, then the current partition is stable under B . If *Unstable* is empty, the partition is stable under all its blocks, and the partition represents the required bisimulation.

As long as some blocks are in *Unstable* (line 3), a single block $B \in \pi$ is taken from this set (line 4) and we split the current partition such that it becomes stable under B . Therefore, we refer to this block as the *splitter*. The set $S' = \{s \in S \mid \exists t \in B. s \rightarrow t\}$ is the reverse image of B (line 6). This set consists of all states that can reach B , and we use S' to define our new blocks. All blocks B' that have a non-empty intersection with S' , i.e., $B' \cap S' \neq \emptyset$, and are not a subset of S' , i.e., $B' \cap S' \neq B'$ (line 7), are split in the subset of states in S' and the subset of states that are not in S' (lines 8–9). These two new blocks are added to the set of *Unstable* blocks (line 10). The number of states is finite, and blocks can be split only a finite number of times. Hence, blocks are only finitely often put in *Unstable*, and so the algorithm terminates.

Algorithm 1: Sequential algorithm based on Kanellakis-Smolka

```

1  $\pi := \pi_0;$ 
2  $Unstable := \pi;$ 
3 while  $Unstable \neq \emptyset$  do
4   foreach  $B \in Unstable$  do
5      $Unstable := Unstable \setminus \{B\};$ 
6      $S' := \{s \in S \mid \exists t \in B. s \rightarrow t\};$ 
7     foreach  $B' \in \pi$  with  $\emptyset \subset B' \cap S' \subset B'$  do
8       // Split  $B'$  into  $B' \cap S'$  and  $B' \setminus S'$ 
9        $\pi := \pi \setminus \{B\};$ 
10       $\pi := \pi \cup \{B' \cap S', B' \setminus S'\};$ 
11       $Unstable := Unstable \cup \{B' \cap S', B' \setminus S'\};$ 
12    end
13  end

```

3.2 The PRAM Algorithm

Next, we describe a PRAM algorithm to solve RCPP that is based on the sequential algorithm given in Algorithm 1.

Block Representation. Given an LTS $A = (S, Act, \rightarrow)$ with $|A| = 1$ and $|S| = n$ states, we assume that the states are labeled with unique indices $0, \dots, n-1$. A partition π in the PRAM algorithm is represented by assigning a block label from a set of block labels L_B to every state. The number of blocks can never be larger than the number of states, hence, we use the indices of the states as block labels: $L_B = S$. We exploit this in the PRAM algorithm to efficiently select a new block label whenever a new block is created. We select the block label of a new block by electing one of its states to be the *leader* of that block and using the index of that state as the block label. By doing so, we maintain an invariant that the leader of a block is also a member of the block.

In a partition π , whenever a block $B \in \pi$ is split into two blocks B' and B'' , the leader s of B which is part of B' becomes the leader of B' , and for B'' , a new state $t \in B''$ is elected to be the leader of this new block. Since the new leader is not part of any other block, the label of t is fresh with respect to the block labels that are used for the other blocks. This method of using state leaders to represent subsets was first proposed in [22, 23].

Data Structures. The common memory contains the following information:

1. $n : \mathbb{N}$, the number of states of the input.
2. $m : \mathbb{N}$, the number of transitions of the input relation.
3. The input, a fixed numbered list of transitions. For every index $0 \leq i < m$ of a transition, a source $source_i \in S$ and target $target_i \in S$ are given, representing the transition $source_i \rightarrow target_i$.

4. $C : L_B \cup \{\perp\}$, the label of the current block that is used as a splitter; \perp indicates that no splitter has been selected.
5. The following is stored in lists of size n , for each state with index i :
 - (a) $mark_i : \mathbb{B}$, a mark indicating whether state i is able to reach the splitter.
 - (b) $block_i : L_B$, the block of which state i is a member.
6. The following is stored in lists of size n , for each potential block with block label i :
 - (a) $new_leader_i : L_B$ the leader of the new block when a split is performed.
 - (b) $unstable_i : \mathbb{B}$ indicating whether π is possibly unstable w.r.t. the block.

As input, we assume that each state with index i has an input variable $I_i \in L_B$ that is the initial block label. In other words, the values of the I_i variables together encode π_0 . Using this input, the initial values of the block label $block_i$ variables are calculated to conform to our block representation with leaders. Furthermore in the initialization, $unstable_i = \text{false}$ for all i that are not used as block label, and true otherwise.

The Algorithm. We provide our first PRAM algorithm in Algorithm 2. The PRAM is started with $\max(n, m)$ processors. These processors are dually used for transitions and states.

The algorithm performs initialisation (lines 1–6), after which each block has selected a new leader (lines 3–4), ensuring that the leader is one of its own states, and the initial blocks are set to unstable. Subsequently, the algorithm enters a single loop that can be explained in three separate parts.

Splitter selection (lines 8–14), executed by n processors. Every variable $mark_i$ is set to false. After this, every processor with index i will check $unstable_i$. If block i is marked unstable the processor tries to write i in the variable C . If multiple write accesses to C happen concurrently in this iteration, then according to the arbitrary PRAM model (see Sect. 2), only one process j will succeed in writing, setting $C := j$ as splitter in this iteration.

Mark states (lines 15–17), executed by m processors. Every processor i is responsible for the transition $s_i \rightarrow t_i$ and checks if t_i ($target_i$) is in the current block C (line 15). If this is the case the processor writes true to $mark_{source_i}$ where $source_i$ is s_i . This mark now indicates that s_i reaches block C .

Performing splits (lines 18–26), executed by n processors. Every processor i compares the mark of state i , i.e., $mark_i$, with the mark of the leader of the block in which state i resides, i.e., $mark_{block_i}$ (line 20). If the marking is different, state i has to be split from $block_i$ into a new block. At line 21, a new leader is elected among the states that form the newly created block. The index of this leader is stored in $new_leader_{block_i}$. The instability of block $block_i$ is set to true (line 22). After that, all involved processors update the block index for their state (line 23) and update the stability of the new block (line 24).

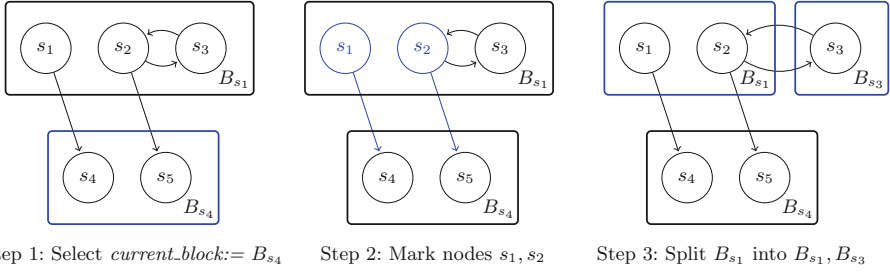


Fig. 1. One iteration of Algorithm 2

The steps of the program are illustrated in Fig. 1. The notation B_{s_i} refers to a block containing all states that have state s_i as their block leader. In the figure on the left, we have two blocks B_{s_1} and B_{s_4} , of which at least B_{s_4} is marked unstable. Block B_{s_4} is selected to be splitter, i.e., $C = B_{s_4}$ at line 12 of Algorithm 2. In the figure in the middle, $mark_i$ is set to true for each state i that can reach B_{s_4} (line 16). Finally, block B_{s_4} is set to stable (line 19), all states compare their mark with the leader’s mark, and the processor working on state s_3 discovers that the mark of s_3 is different from the mark of s_1 , so s_3 is elected as leader of the new block B_{s_3} at line 21 of Algorithm 2. Both B_{s_1} and B_{s_3} are set to unstable (lines 22 and 24).

The algorithm repeats execution of the **while**-loop until all blocks are marked stable.

3.3 Correctness

The $block_i$ list in the common memory at the start of iteration k defines a partition π_k where states $s \in S$ with equal block labels $block_i$ form the blocks:

$$\pi_k = \{\{s \in S \mid block_s = s'\} \mid s' \in S\} \setminus \{\emptyset\}$$

A run of the program produces a sequence π_0, π_1, \dots of partitions. Partition π_k is a refinement of every partition $\pi_0, \pi_1, \dots, \pi_{k-1}$, since blocks are only split and never merged.

A partition π induces a relation of which the blocks are the equivalence classes. For an input partition π_0 we call the relation induced by the coarsest refinement of π_0 that is a bisimulation relation \Leftrightarrow_{π_0} .

We now prove that Algorithm 2 indeed solves RCPP. We first introduce Lemma 1 which is invariant throughout the execution and expresses that states which are related by \Leftrightarrow_{π_0} are never split into different blocks. This lemma implies that if a refinement forms a bisimulation relation, it is the coarsest.

Lemma 1. *Let S be the input set of states, $\rightarrow: S \times S$ the input relation and π_0 the input partition. Let π_1, π_2, \dots be the sequence of partitions produced by*

Algorithm 2: The algorithm for RCPP for each processor P_i in the PRAM

```

1 if  $i < n$  then
2    $unstable_i := \text{false};$ 
3    $new\_leader_{I_i} := i;$ 
4    $block_i := new\_leader_{I_i};$ 
5    $unstable_{block_i} := \text{true};$ 
6 end
7 do
8    $C := \perp;$ 
9   if  $i < n$  then
10     $mark_i := \text{false};$ 
11    if  $unstable_i$  then
12       $C := i;$ 
13    end
14  end
15  if  $i < m$  and  $block_{target_i} = C$  then
16     $mark_{source_i} := \text{true};$ 
17  end
18  if  $i < n$  and  $C \neq \perp$  then
19     $unstable_C := \text{false};$ 
20    if  $mark_i \neq mark_{block_i}$  then
21       $new\_leader_{block_i} := i;$ 
22       $unstable_{block_i} := \text{true};$ 
23       $block_i := new\_leader_{block_i};$ 
24       $unstable_{block_i} := \text{true};$ 
25    end
26  end
27 while  $C \neq \perp;$ 

```

Algorithm 2, then for all initial blocks $B \in \pi_0$, states $s, t \in B$ and iterations $k \in \mathbb{N}$:

$$s \Leftrightarrow_{\pi_0} t \implies \exists B \in \pi_k. s, t \in B$$

Proof. This is proven by induction on k . In the base case, π_0 , this is true by default. Now assume for a particular $k \in \mathbb{N}$ that the property holds. We know that the partition π_{k+1} is obtained by splitting with respect to a block $C \in \pi_k$. For two states $s, t \in S$ with $s \Leftrightarrow_{\pi_0} t$ we know that s and t are in the same block in π_k . In the case that both s and t do not reach C , then $mark_s = mark_t = \text{false}$. Since they were in the same block, they will be in the same block in π_{k+1} .

Now consider the case that at least one of the states is able to reach C . Without loss of generality say that s is able to reach C . Then there is a transition $s \rightarrow s'$ with $s' \in C$. By Definition 2, there exists a $t' \in S$ such that $t \rightarrow t'$ and $s' \Leftrightarrow_{\pi_0} t'$. By the induction hypothesis we know that since $s' \Leftrightarrow_{\pi_0} t'$, s' and t' must be in the same block in π_k , i.e., t' is in C . This witnesses that t is also able to reach C and we must have $mark_s = mark_t = \text{true}$. Since the states s and t

are both marked and are in the same block in π_k , they will remain in the same block in π_{k+1} .

Lemma 2. *Let S be the input set of states with $\rightarrow: S \times S$, $L_B = S$ the block labels, and π_n the partition stored in the memory after the termination of Algorithm 2. Then the relation induced by π_n is a bisimulation relation.*

Proof. Since the program finished, we know that for all block indices $i \in L_B$ we have $unstable_i = \text{false}$. For a block index $i \in L_B$, $unstable_i$ is set to false if the partition π_k , after iteration k , is stable under the block with index i and set to true if it is split. So, by Fact 1, we know π_n is stable under every block B , hence stable. Next, we prove that a stable partition is a bisimulation relation.

We show that the relation R induced by π_n is a bisimulation relation. Assume states $s, t \in S$ with sRt are in block $B \in \pi_n$. Consider a transition $s \rightarrow s'$ with $s' \in S$. State s' is in some block $B' \in \pi_n$, and since the partition is stable under block B' , and s is able to reach B' , by the definition of stability, we know that t is also able to reach B' . Therefore, there must be a state $t' \in B'$ such that $t \rightarrow t'$ and $s'Rt'$. Finally, by the fact that R is an equivalence relation we know that R is also symmetric, therefore it is a bisimulation relation.

Theorem 1. *The partition resulting from executing Algorithm 2 forms the coarsest relational partition for a set of states S and a transition relation $\rightarrow: S \times S$, solving RCPP.*

Proof. By Lemma 2, the resulting partition is a bisimulation relation. Lemma 1 implies that it is the coarsest refinement which is a bisimulation.

3.4 Complexity Analysis

Every step in the body of the **while**-loop can be executed in constant time. So the asymptotic complexity of this algorithm is given by the number of iterations.

Theorem 2. *RCPP on an input with m transitions and n states is solved by Algorithm 2 in $\mathcal{O}(n)$ time using $\max(n, m)$ CRCW PRAM processors.*

Proof. In iteration $k \in \mathbb{N}$ of the algorithm, let us call the total number of blocks $N_k \in \mathbb{N}$ and the number of unstable blocks $U_k \in \mathbb{N}$. Initially, $N_0 = U_0 = |\pi_0|$. In every iteration k , a number of blocks $l_k \in \mathbb{N}$ is split, resulting in l_k new blocks, so the new total number of blocks at the end of iteration k is $N_{k+1} = N_k + l_k$.

First the current block C in iteration k which was unstable is set to stable which causes the number of unstable blocks to decrease by one. In this iteration k the l_k blocks B_1, \dots, B_{l_k} are split, resulting in l_k newly created blocks. These l_k blocks are all unstable. A number of blocks $l'_k \leq l_k$ of the blocks B_1, \dots, B_{l_k} , were stable and are set to unstable again. The block C which was set to stable is possibly one of these l'_k blocks which were stable and set to unstable again. The total number of unstable blocks at the end of iteration k is $U_{k+1} = U_k + l_k + l'_k - 1$.

For all $k \in \mathbb{N}$, in iteration k we calculate the total number of blocks $N_k = |\pi_0| + \sum_{i=0}^{k-1} (l_i)$ and unstable blocks $U_k = |\pi_0| - k + \sum_{i=0}^{k-1} (l_i + l'_i)$. The number

of iterations is given by $k = \sum_{i=0}^{k-1} (l_i + l'_i) - U_k + |\pi_0|$. By definition, $l'_i \leq l_i$, and the total number of newly created blocks is $\sum_{i=0}^{k-1} (l_i) = N_k - |\pi_0|$, hence $\sum_{i=0}^{k-1} (l_i + l'_i) \leq 2 \sum_{i=0}^{k-1} (l_i) \leq 2N_k - 2|\pi_0|$. The number of unstable blocks is always positive, i.e., $U_k \geq 0$, and the total number of blocks can never be larger than the number of states, i.e., $N_k \leq n$, so the total number of iterations z is bounded by $z \leq 2N_z - |\pi_0| \leq 2n - |\pi_0|$.

4 Bisimulation Coarsest Refinement Problem

In this section we extend our algorithm to the Bisimulation Coarsest Refinement Problem (BCRP), i.e., to LTSs with multiple action labels.

Solving BCRP can in principle be done by translating an LTS to a Kripke structure, for instance by using the method described in [18]. This translation introduces a new state for every transition, resulting in a Kripke structure with $n + m$ states. If the number of transitions is significantly larger than the number of states, then the number of iterations of our algorithm increases undesirably.

4.1 The PRAM Algorithm

Instead of introducing more states, we introduce multiple marks per state, but in total we have no more than m marks. For each state s , we use a mark variable for each different outgoing action label relevant for s , i.e., for each a for which there is a transition $s \xrightarrow{a} t$ to some state t . Each state may have a different set of outgoing action labels and thus a different set of marks. Therefore, we first perform a preprocessing procedure in which we group together states that have the same set of outgoing action labels. This is valid, since two bisimilar states must have the same outgoing actions. That two states of the same block have the same set of action labels is then an invariant of the algorithm, since in the sequence of produced partitions, each partition is a refinement of the previous one. For the extended algorithm, we need to maintain extra information in addition to the information needed for Algorithm 2. For an input LTS $A = (S, Act, \rightarrow)$ with n states and m transitions the extra information is:

1. Each action label has an index $a \in \{0, \dots, |Act| - 1\}$.
2. The following is stored in lists of size m , for each transition $s \xrightarrow{a} t$ with transition index $i \in \{0, \dots, m - 1\}$:
 - (a) $a_i := a$
 - (b) $order_i : \mathbb{N}$, the order of this action label, with respect to the source state s . E.g., if a state s has the list $[1, 3, 6]$ of outgoing action labels, and transition i has label 3, then $order_i$ is 1 (we start counting from 0).
3. $mark : [\mathbb{B}]$, a list of up to m marks, in which there is a mark for every state, action pair for which it holds that the state has at least one outgoing transition labelled with that action. This list can be interpreted as the concatenation of sublists, where each sublist contains all the marks for one state. For each state $s \in S$ we have:

- (a) $off(s) : \mathbb{N}$, the offset to access the beginning of the sublist of the marks of the state s in $mark$. The positions $mark_{off(s)}$ up to $mark_{off(s+1)}$ contain the sublist of marks for state s . E.g., if state s has outgoing transitions with 3 distinct action labels, we know that $off(s+1) \equiv off(s) + 3$, and we have 3 marks for state s . We write $mark_{off(s)+order_i}$ to access the mark for transition i which has source state s .
4. $mark_length$: The length of the mark list. This allows us to reset all marks in constant time using $mark_length$ processors. This number is not larger than the number of transitions ($mark_length \leq m$).
5. In a list of size n , we store for each state $s \in S$ a variable $split_s : \mathbb{B}$. This indicates if the state will be split off from its block.

With this extra information, we can alter Algorithm 2 to work with labels. The new version is given in Algorithm 3. The changes involve the following:

1. Lines 7–9: Reset the $mark$ list.
2. Line 11: Reset the $split$ list.
3. Line 17: When marking the transitions, we do this for the correct action label, using $order_i$. Note the indexing into $mark$. It involves the offset for the state $source_i$, and $order_i$.
4. Lines 19–21: We tag a state to be split when it differs for any action from the block leader.
5. Line 24: If a state was tagged to be split in the previous step, it should split from its leader.
6. Line 29: If any block was split, the partition may not be stable w.r.t. the splitter.

To use Algorithm 3, we need to do two preprocessing steps. First, we need to partition the states w.r.t. their set of outgoing action labels. This can be done with an altered version of Algorithm 2, which does one iteration for each action label. For the second preprocessing step, we need to gather the extra information that is needed in Algorithm 3. This is done via sorting the action labels and subsequently performing some parallel segmented (prefix) sums [19]. In total the preprocessing takes $\mathcal{O}(|Act| + \log m)$ time. For details how this is implemented see the full version of this paper [14].

4.2 Complexity and Correctness

For Algorithm 3, we need to prove why it takes a linear number of steps to construct the final partition. This is subtle, as an iteration of the algorithm does not necessarily produce a stable block.

Theorem 3. *Algorithm 3 on an input LTS with n states and m transitions will terminate in $\mathcal{O}(n + |Act|)$ steps.*

Proof. The total preprocessing takes $\mathcal{O}(|Act| + \log m)$ steps, after which the **while**-loop will be executed on a partitioning π_0 which was the result of the

Algorithm 3: The algorithm for BCRP, the highlighted lines differ from Algorithm 2.

```

1  if  $i < n$  then
2    |    $unstable_i := \text{false};$ 
3    |    $unstable_{block_i} := \text{true};$ 
4  end
5  do
6    |    $C := \perp;$ 
7    |   if  $i < \text{mark\_length}$  then
8    |   |    $mark_i := \text{false};$ 
9    |   end
10   |   if  $i < n$  then
11   |   |    $split_i := \text{false};$ 
12   |   |   if  $unstable_i$  then
13   |   |   |    $C := i;$ 
14   |   |   end
15   |   end
16   |   if  $i < m$  and  $block_{target_i} = C$  then
17   |   |    $mark_{off(source_i)+order_i} := \text{true};$ 
18   |   end
19   |   if  $i < m$  and  $mark_{off(source_i)+order_i} \neq mark_{off(block_{source_i})+order_i}$  then
20   |   |    $split_{source_i} := \text{true};$ 
21   |   end
22   |   if  $i < n$  &  $C \neq \perp$  then
23   |   |    $unstable_C := \text{false};$ 
24   |   |   if  $split_i$  then
25   |   |   |    $new\_leader_{block_i} := i;$ 
26   |   |   |    $unstable_{block_i} := \text{true};$ 
27   |   |   |    $block_i := new\_leader_{block_i};$ 
28   |   |   |    $unstable_{block_i} := \text{true};$ 
29   |   |   |    $unstable_C := \text{true};$ 
30   |   |   end
31   |   end
32 while  $C \neq \perp;$ 

```

preprocessing on the partition $\{S\}$. Every iteration of the **while**-loop is still executed in constant time. Using the structure of the proof of Theorem 2, we derive a bound on the number of iterations.

At the start of iteration $k \in \mathbb{N}$ the total number of blocks and unstable blocks are $N_k, U_k \in \mathbb{N}$, initially $U_0 = N_0 = |\pi_0|$. In iteration k , a number l_k of blocks is split in two blocks, resulting in l_k new blocks, meaning that $N_{k+1} = N_k + l_k$. All new l_k blocks are unstable and a number $l'_k \leq l_k$ of the old blocks that are split were stable at the start of iteration k and are now unstable. If $l_k = l'_k = 0$ there are no blocks split and the current block C becomes stable. We indicate this with a variable c_k : $c_k = 1$ if $l_k = 0$, and $c_k = 0$, otherwise. The total number

of iterations up to iteration k in which no block is split is given by $\sum_{i=0}^{k-1} c_i$. The number of iterations in which at least one block is split is given by $k - \sum_{i=0}^{k-1} c_i$.

If in an iteration k at least one block is split, the total number of blocks at the end of iteration k is strictly higher than at the beginning, hence for all $k \in \mathbb{N}$, $N_k \geq k - \sum_{i=0}^{k-1} c_i$. Hence, $N_k + \sum_{i=0}^{k-1} c_i$ is an upper bound for k .

We derive an upper bound for the number of iterations in which no blocks are split using the total number of unstable blocks. In iteration k there are $U_k = \sum_{i=0}^{k-1} (l_i + l'_i) - \sum_{i=0}^{k-1} c_i + |\pi_0|$ unstable blocks. Since the sum of newly created blocks $\sum_{i=0}^{k-1} (l_i) = N_k - |\pi_0|$ and $l'_i \leq l_i$, the number of unstable blocks U_k is bounded by $2N_k - \sum_{i=0}^{k-1} c_i - |\pi_0|$. Since $U_k \geq 0$ we have the bound $\sum_{i=0}^{k-1} c_i \leq 2N_k - |\pi_0|$. This gives the bound on the total number of iterations $z \leq 3N_z - |\pi_0| \leq 3n - |\pi_0|$.

With the time for preprocessing this makes the run time complexity $\mathcal{O}(n + |Act| + \log m)$. Since the number of transitions m is bounded by $|Act| \times n^2$, this simplifies to $\mathcal{O}(n + |Act|)$.

5 Experimental Results

In this section we discuss the results of our implementation of Algorithm 3 from Sect. 4. Note that this implementation is not optimized for the specific hardware it runs on, since the goal of this paper is to provide a generic parallel algorithm. This implementation is purely a proof of concept, to show that our algorithm can be mapped to contemporary hardware and to understand how the algorithm scales with the size of the input.

The implementation targets GPUs since a GPU closely resembles a PRAM and supports a large amount of parallelism. The algorithm is implemented in CUDA version 11.1 with use of the Thrust library.¹ As input, we chose all benchmarks of the VLTS benchmark suite² for which the implementation produced a result within 10 min. The VLTS benchmarks are LTSs that have been derived from real concurrent system models.

The experiments were run on an NVIDIA Titan RTX with 24 GB memory and 72 Streaming Multiprocessors, each supporting up to 1,024 threads in flight. Although this GPU supports 73,728 threads in flight, it is very common to launch a GPU program with one or even several orders of magnitude more threads, in particular to achieve load balancing between the Streaming Multiprocessors and to hide memory latencies. In fact, the performance of a GPU program usually relies on that many threads being launched.

The left-hand side of Table 1 shows the results of the experiments we conducted. The $|Act|$ column corresponds to the number of different action labels. The $|Blocks|$ column indicates the number of different blocks at the end of the algorithm, where each block contains only bisimilar states. With $\#It$ we refer to the number of **while**-loop iterations that were executed (see Algorithm 3),

¹ The source code can be found at <https://github.com/sakehl/gpu-bisimulation>.

² <https://cadp.inria.fr/resources/vlts/>.

Table 1. Benchmark results for Par-BCRP (Algorithm 3) on a GPU, times (T) are in ms. The right-hand side compares the total times from the different algorithms.

Benchmark name	Act	Blocks	#It	T_{pre}	T_{alg}	#It/n	#It/ Blocks	$T_{Par-BCRP/n}$	$T_{alg}/\#It$	$T_{Par-BCRP}$	T_{LR}	T_{Wss}	T_{Wms}
Vasy_0.1	2	9	16	0.50	0.37	0.06	1.78	0.003	0.023	0.87	2.29	0.49	0.45
Cwi_1.2	26	1,132	2,786	0.63	56.5	1.43	2.46	0.029	0.020	57.1	17	18.8	21.8
Vasy_1.4	6	28	45	0.56	1.01	0.04	1.61	0.001	0.022	1.58	4.78	1.68	0.62
Cwi_3.14	2	62	122	0.63	2.68	0.03	1.97	0.001	0.022	3.30	60	3.80	3.72
Vasy_5.9	31	145	193	0.84	4.22	0.04	1.33	0.001	0.022	5.06	134	35.3	3.45
Vasy_8.24	11	416	664	0.70	13.9	0.07	1.59	0.002	0.021	15	277	31.5	3.03
Vasy_8.38	81	219	319	1.12	6.64	0.04	1.46	0.001	0.021	7.76	127	35.1	5.94
Vasy_10.56	12	2,112	3,970	0.73	82.0	0.37	1.88	0.008	0.021	82.7	860	40.9	4.6(0.2)
Vasy_18.73	17	4,087	6,882	1.01	142	0.37	1.68	0.008	0.021	143	1,354	211	21.7
Vasy_25.25	25,216	25,217	25,218	159	519	1.00	1.00	0.027	0.021	678	21,960	t.o	416
Vasy_40.60	3	40,006	87,823	0.87	1,810	2.20	2.20	0.045	0.021	1,811	17,710	1,290	1,230
Vasy_52.318	17	8,142	15,985	2.52	338	0.31	1.96	0.007	0.021	340	11,855	368	152(20)
Vasy_65.2621	72	65,536	98,730	12.2	10,050	1.51	1.51	0.154	0.102	10,060	t.o	27,000	1,230
Vasy_66.1302	81	66,929	91,120	6.00	5,745	1.36	1.36	0.086	0.063	5,752	480,600	20,450	240(20)
Vasy_69.520	135	69,754	113,246	4.13	3,780	1.62	1.62	0.054	0.033	3,780	94,800	16,090	35.4
Vasy_83.325	211	83,436	148,012	4.41	3,093	1.77	1.77	0.037	0.021	3,097	57,190	21,500	5,880
Vasy_116.368	21	116,456	210,537	2.50	5,900	1.81	1.81	0.051	0.028	5,900	80,900	6,360	2,930
Cwi_142.925	7	3,410	5,118	4.85	238	0.04	1.50	0.002	0.047	243	3,363	220(30)	140(20)
Vasy_157.297	235	4,289	9,682	4.58	201	0.06	2.26	0.001	0.021	206	1,058	1,240	579
Vasy_164.1619	37	1,136	1,630	8.34	125	0.01	1.43	0.001	0.077	134	8,173	470(30)	46.8
Vasy_166.651	211	83,436	145,029	6.13	5,710	0.87	1.74	0.034	0.039	5,720	80,210	29,660	9,560
Cwi_214.684	5	77,292	149,198	3.58	6,948	0.70	1.93	0.032	0.047	6,952	19,250	440(30)	450(50)
Cwi_371.641	61	33,994	85,858	4.72	4,050	0.23	2.53	0.011	0.047	4,050	26,940	6,970	1,548
Vasy_386.1171	73	113	199	7.38	14.0	0.00	1.76	0.000	0.070	21	334	30.6	34.8
Cwi_566.3984	11	15,518	23,774	16.0	3,707	0.04	1.53	0.007	0.156	3,723	98,200	6,700	2,200(200)
Vasy_574.13561	141	3,577	5,860	71.5	3,770	0.01	1.64	0.007	0.643	3,841	144,810	11,700	1,853
Vasy_720.390	49	3,292	3,782	3.97	143	0.01	1.15	0.0002	0.038	147	2,454	1,633	183
Vasy_1112.5290	23	265	365	24.0	99.3	0.0003	1.38	0.0001	0.272	123	4,570	293	36.8
Cwi_2165.8723	26	31,906	66,132	37.0	23,660	0.03	2.07	0.011	0.358	23,700	140,170	9,700	1,965
Cwi_2416.17605	15	95,610	152,099	64.1	96,400	0.06	1.59	0.040	0.634	96,500	257,200	16,300(1100)	15,300
Vasy_6020.19353	511	7,168	12,262	221	11,690	0.002	1.71	0.002	0.954	11,910	107,900	34,000(2000)	19,230
Vasy_6120.11031	125	5,199	10,014	74.0	6,763	0.002	1.93	0.001	0.675	6,837	55,750	7,010	1,280
Vasy_8082.42933	211	408	660	281	1,149	0.0001	1.62	0.0002	1.739	1,429	17,272	5,530	2,030

before all blocks became stable. The number of states and transitions can be derived from the benchmark name. In the benchmark ‘ $X_N M$ ’, $N * 1000$ is the number states and $M * 1000$ is the number of transitions. The T_{pre} give the preprocessing times in milliseconds, which includes doing the memory transfers to the GPU, sorting the transitions and partitioning. The T_{alg} give the times of the core algorithm, in milliseconds. The $T_{Par-BCRP}$ is the sum of the preprocessing and the algorithm, in milliseconds. We have not included the loading times for the files and the first CUDA API call that initializes the device. We ran each benchmark 10 times and took the averages. The standard deviation of the total times varied between 0% and 3% of the average, thus 10 runs are sufficient. All the times are rounded with respect to the standard error of the mean.

We see that the bound as proven in Sect. 4.2 ($k \leq 3n$) is indeed respected, $\#It/n$ is at most 2.20, and most of the time below that. The number of iterations is tightly related to the number of blocks that the final partition has, the $\#It/|Blocks|$ column varies between 1.00 and 2.53. This can be understood by the fact that each iteration either splits one or more blocks or marks a block as stable, and all blocks must be checked on stability at least once. This also means that for certain LTSs the algorithm scales better than linearly in n . The preprocessing often takes the same amount of time (about a few milliseconds). Exceptions are those cases with a large number of actions and/or transitions.

Concerning the run times, it is not true that each iteration takes the same amount of time. A GPU is not a perfect PRAM machine. There are two key differences. Firstly, we suspect that the algorithm is memory bound since it is performing a limited amount of computations. The memory accesses are irregular, i.e., random, which caches can partially compensate, but for sufficiently large n and m , the caches cannot contain all the data. This means that as the LTSs become larger, memory accesses become relatively slower. Secondly, at a certain moment, the maximum number of threads that a GPU can run in parallel is achieved, and adding more threads will mean more run time. These two effects can best be seen in the $T_{alg}/\#It$ column, which corresponds to the time per iteration. The values are around 0.02 up to 300,000 transitions, but for a larger number of states and transitions, the amount of time per iteration increases.

5.1 Experimental Comparison

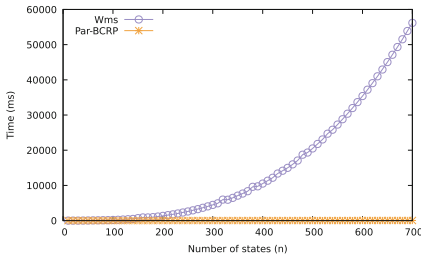
We compared our implementation (Par-BCRP) with an implementation of the algorithm by Lee and Rajasekaran (LR) [12] on GPUs, and the optimized GPU implementation by Wijts based on *signature-based* bisimilarity checking [2], with *multi-way splitting* (Wms) and with *single-way splitting* (Wss) [22]. Multi-way splitting indicates that a block is split in multiple blocks at once, which is achieved by computing a signature for each state in every partition refinement iteration, and splitting each block off into sets of states, each containing all the states with the same signature. The signature of a state is derived from the labels of the blocks that this state can reach in the current partition. Note that we are not including comparisons with CPU bisimulation checking tools; the fact that those tools run on completely different hardware makes a comparison problematic, and such a comparison does not serve the purpose of evaluating the feasibility of implementing Algorithm 3. Optimising our implementation to make it competitive with CPU tools is planned for future work.

The running times of the different algorithms can be found in the right-hand side of Table 1. Similarly to our previous benchmarks, the algorithms were run 10 times on the same machine using the same VLTS benchmark suite with a timeout of 10 min. In some cases, the non-deterministic behaviour of the algorithms Wms and Wss led to high variations in the runs. In cases where the standard error of the mean was more than 5% of the mean value, we have added the standard error in Table 1 in between parentheses. Furthermore, all the results are rounded with respect to the standard error of the mean. As a pre-processing step for the LR, Wms and Wss algorithms the input LTSs need to be sorted. We did not include this in the times, nor the reading of files and the first CUDA API call (which initializes the GPU).

This comparison confirms the expectation that our algorithm in all cases (except one small LTS) outperforms LR. This confirms our expectation that LR is not suitable for massive parallel devices such as GPUs.

Furthermore, the comparison demonstrates that in most cases our algorithm (Par-BCRP) outperforms Wss. In some benchmarks (Cwi_1_2, Cwi_214_684, Cwi_2165_8723 and Cwi_2416_17605) Wss is more than twice as fast, but in 16

other cases our algorithm is more than twice as fast. The last comparison shows us that our algorithm does not out-perform Wms. Wms employs multi-way splitting which is known to be very effective in practice. Furthermore, contrary to our implementation, Wms is optimized for GPUs while the focus of the current work is to improve the theoretical bounds and describe a general algorithm.



States	Run time Wms (ms)	Par-BCRP (ms)
10	1	1
100	182	5
200	1350	9
300	4463	13
400	10519	18
500	20508	22
600	35392	26
700	56183	30

Fig. 2. Run times of Par-BCRP and Wms on the LTS Fan_out_n .

In order to understand the difference between Wms and our algorithm better, we analysed the complexity of Wms [22]. In general this algorithm is quadratic in time, and the linearity claim in [22] depends on the assumption that the fan-out of ‘practical’ transition systems is bounded, i.e., every state has no more than c outgoing transitions for c a (low) constant. We designed the transition systems Fan_out_n for $n \in \mathbb{N}^+$ to illustrate the difference. The LTS $Fan_out_n = (S, \{a, b\}, \rightarrow)$ has n states: $S = \{0, \dots, n - 1\}$. The transition function contains $i \xrightarrow{a} i + 1$ for all states $1 < i < n - 1$. Additionally, from state 0 and 1 there are transitions to every state: $0 \xrightarrow{b} i, 1 \xrightarrow{b} i$ for all $i \in S$. This LTS has n states, $3n - 3$ transitions and a maximum out degree of n transitions.

Figure 2 shows the results of calculating the bisimulation equivalence classes for Fan_out_n , with Wms and Par-BCRP. It is clear that the run time for Wms increases quadratically as the number of states grows linearly, already becoming untenable for a small amount of states. On the other hand, in conformance with our analysis, our algorithm scales linearly.

6 Conclusion

We proposed and implemented an algorithm for RCPP and BCRP. We proved that the algorithm stops in $\mathcal{O}(n + |Act|)$ steps on $\max(n, m)$ CRCW PRAM processors. We implemented the algorithm for BCRP in CUDA, and conducted experiments that show the potential to compute bisimulation in practice in linear time. Further advances in parallel hardware will make this more feasible.

For future work, it is interesting to investigate whether RCPP can be solved in sublinear time, that is $\mathcal{O}(n^\epsilon)$ for a $\epsilon < 1$, as requested in [12]. It is also intriguing whether the practical effectiveness of the algorithm in [22] by splitting blocks

simultaneously can be combined with our algorithm, while preserving the linear time upperbound. Furthermore, it remains an open question whether our algorithm can be generalised for weaker bisimulations, such as weak and branching bisimulation [7,9]. The main challenge here is that the transitive closure of so-called internal steps needs to be taken into account.

References

1. Balcázar, J., Gabarro, J., Santha, M.: Deciding bisimilarity is P-complete. *Formal Aspects Comput.* **4**(1), 638–648 (1992). <https://doi.org/10.1007/BF03180566>
2. Blom, S., Orzan, S.: Distributed branching bisimulation reduction of state spaces. *Electr. Notes Theoret. Comput. Sci.* **89**(1), 99–113 (2003). [https://doi.org/10.1016/S1571-0661\(05\)80099-4](https://doi.org/10.1016/S1571-0661(05)80099-4)
3. Blom, S., van de Pol, J.: Distributed branching bisimulation minimization by inductive signatures. In: Brim, L., van de Pol, J. (eds.) *Proceedings 8th International Workshop on Parallel and Distributed Methods in verification, PDMC 2009, Eindhoven, The Netherlands, 4th November 2009, EPTCS*, vol. 14, pp. 32–46 (2009). <https://doi.org/10.4204/EPTCS.14.3>
4. Bunte, O., et al.: The mCRL2 toolset for analysing concurrent systems. In: Vojnar, T., Zhang, L. (eds.) *TACAS 2019. LNCS*, vol. 11428, pp. 21–39. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17465-1_2
5. De Nicola, R., Vaandrager, F.: Action versus state based logics for transition systems. In: Guessarian, I. (ed.) *LITP 1990. LNCS*, vol. 469, pp. 407–419. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-53479-2_17
6. Fortune, S., Wyllie, J.: Parallelism in random access machines. In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pp. 114–118 (1978). <https://doi.org/10.1145/800133.804339>
7. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. *J. ACM* **43**(3), 555–600 (1996). <https://doi.org/10.1145/233551.233556>
8. Groote, J.F., Wijs, A.: An $O(m \log n)$ algorithm for stuttering equivalence and branching bisimulation. In: Chechik, M., Raskin, J.-F. (eds.) *TACAS 2016. LNCS*, vol. 9636, pp. 607–624. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_40
9. Jansen, D.N., Groote, J.F., Keiren, J.J.A., Wijs, A.: An $O(m \log n)$ algorithm for branching bisimilarity on labelled transition systems. In: *TACAS 2020. LNCS*, vol. 12079, pp. 3–20. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45237-7_1
10. Jeong, C., Kim, Y., Oh, Y., Kim, H.: A faster parallel implementation of Kanellakis-Smolka algorithm for bisimilarity checking. In: *Proceedings of the International Computer Symposium*. Citeseer (1998)
11. Kanellakis, P., Smolka, S.: CCS expressions, finite state processes, and three problems of equivalence. *Inf. Comput.* **86**(1), 43–68 (1990). [https://doi.org/10.1016/0890-5401\(90\)90025-D](https://doi.org/10.1016/0890-5401(90)90025-D)
12. Lee, I., Rajasekaran, S.: A parallel algorithm for relational coarsest partition problems and its implementation. In: Dill, D.L. (ed.) *CAV 1994. LNCS*, vol. 818, pp. 404–414. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58179-0_71
13. Leiserson, C.E., et al.: There’s plenty of room at the top: what will drive computer performance after Moore’s law? *Science* **368**(6495) (2020). <https://doi.org/10.1126/science.aam9744>

14. Martens, J., Groote, J., Haak, L.v.d., Hijma, P., Wijs, A.: A linear parallel algorithm to compute bisimulation and relational coarsest partitions. arXiv preprint [arXiv:2105.11788](https://arxiv.org/abs/2105.11788) (2021)
15. Milner, R. (ed.): A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980). <https://doi.org/10.1007/3-540-10235-3>
16. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **16**(6), 973–989 (1987). <https://doi.org/10.1137/0216062>
17. Rajasekaran, S., Lee, I.: Parallel algorithms for relational coarsest partition problems. IEEE Trans. Parallel Distrib. Syst. **9**(7), 687–699 (1998). <https://doi.org/10.1109/71.707548>
18. Reniers, M.A., Schoren, R., Willemse, T.: Results on embeddings between state-based and event-based systems. Comput. J. **57**(1), 73–92 (2014). <https://doi.org/10.1093/comjnl/bxs156>
19. Sengupta, S., Harris, M., Garland, M., Owens, J.: Efficient parallel scan algorithms for GPUs. In: Scientific Computing with Multicore and Accelerators, chap. 19, pp. 413–442. Taylor & Francis (2011)
20. Stockmeyer, L., Vishkin, U.: Simulation of parallel random access machines by circuits. SIAM J. Comput. **13**(2), 409–422 (1984). <https://doi.org/10.1137/0213027>
21. Valmari, A.: Simple bisimilarity minimization in $O(m \log n)$ time. Fundam. Informaticae **105**(3), 319–339 (2010). <https://doi.org/10.3233/FI-2010-369>
22. Wijs, A.: GPU accelerated strong and branching bisimilarity checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 368–383. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_29
23. Wijs, A., Katoen, J.-P., Bošnački, D.: Efficient GPU algorithms for parallel decomposition of graphs into strongly connected and maximal end components. Formal Methods Syst. Des. **48**(3), 274–300 (2016). <https://doi.org/10.1007/s10703-016-0246-7>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.





Automated Generation of Initial Configurations for Testing Component Systems

Frédéric Dadeau^(✉), Jean-Philippe Gros, and Olga Kouchnarenko

Univ. Bourgogne Franche-Comté, CNRS, FEMTO-ST Institute,
15B avenue des Montboucons, 25030 Besançon, Cedex, France
`{frederic.dadeau,jean-philippe.gros,olga.kouchnarenko}@femto-st.fr`

Abstract. In the context of component-based systems, this paper presents the automated generation of initial states, from which an adaptive system starts to receive sequences of events that aim to provoke reconfigurations. For generating these states, also called configurations, we present a combinatorial algorithm supporting various architectural elements and relationships among them, while satisfying consistency constraints expressed by invariants. Moreover, this algorithm deals with the system-dependant instantiations of the primitive and composite components, parameters and relations, in order to produce meaningful structured configurations. While testing adaptation policies for component-based systems, this algorithm allows us to improve the capability of fulfilling coverage criteria by using different initial configurations. To illustrate the approach, the paper reports on experiments on a simulation with platoons of autonomous vehicles.

1 Introduction

Even if models of component-based systems are very heterogeneous, most of them consider software components either as black boxes, or as grey boxes if some of their inner features are visible, having fully-described interfaces. Systems' behaviour is then specified using components' definitions. In general, the system state, also called a configuration, is the specific definition of the elements that define what a system is composed of, while a reconfiguration can be seen as a transition from a configuration to another. In this context, adaptation rules or policies can be used to guide dynamic reconfigurations of component-based systems [6, 10, 18] by using either architectural constraints on configurations, or events, or temporal constraints over sequences of events and reconfigurations.

Overall, our goal is to validate that the adaptation policy rules are faithfully implemented by the system. In [7], the system execution has been validated w.r.t. the adaptation policy by checking that the reconfigurations that are triggered during the execution correspond to those authorized in the adaptation policy. In addition, [8] addresses the issue of validating that the system execution, which starts from a particular configuration, respects the utilities of the reconfiguration

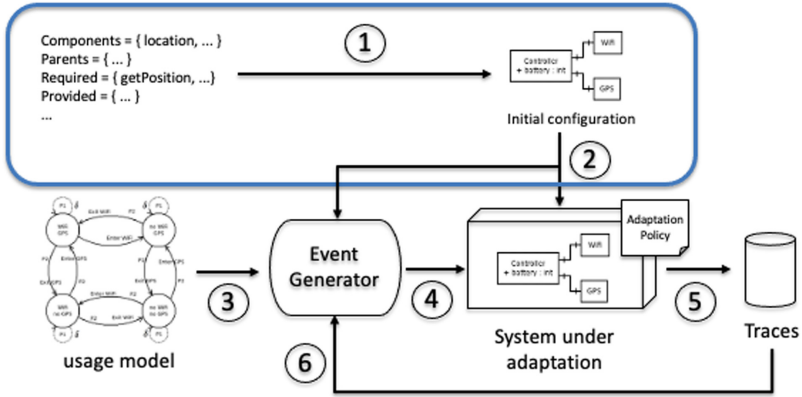


Fig. 1. The process of online test generation for adaptive systems

rules of the adaptation policy. It is easy to see that the testing process depends on initial configurations. For example, in the context of autonomous systems, when the execution starts from a configuration with the full battery, only a few reconfigurations are expected and, consequently, the test cases to cover aimed behaviour with reconfigurations may be long enough. On the contrary, starting from a configuration where the battery level is low, provokes reconfigurations to save energy, and the test cases are expected to be shorter. So, to go further, the present paper presents the automated generation of initial configurations, from which the adaptive system starts to receive sequences of events that aim to provoke reconfigurations.

This process is summarized in Fig. 1. The adaptive system’s architecture is described by a component-based model, from which initial configurations are automatically generated while instantiating the model (1). These initial configurations are necessary inputs to initialize testing (2) of the system, as the tests are executed starting from them, as well as the system itself. To take into account the environment in which the system is executed, usage models [25] for components are provided as inputs as well. Starting from initial configurations, test cases are composed of the events, which are extracted from components usage models (3). These events are sent by the test generator to the system (4), in an online testing manner: as the system’s behavior depends on the environment, test outputs are observed on the reconfiguration trace (5), and analyzed by the test generator (6) to both guide the next event to be sent to the system under test (4), and verify that the system behaves as expected w.r.t. the various artifacts that are available, namely adaptation policies with temporal properties (not shown in Fig. 1). This last point has been described in [7], where as the present paper focuses on points (1) and (2).

In order to generate initial configurations, we first considered a random data generator. However, due to the complexity of the structures to be generated for component-based systems, such a random generator could possibly not

terminate, or hardly converge to a relevant configuration, which is realistic in terms of architecture. Indeed, the relationships that are defined by the component model, regarding parenting relationships, delegations and bindings, define a constraint satisfaction problem (CSP), that cannot be effectively solved by a random process. While constraint solvers exist, such as CLPS [5], using them is problematic in our context. First issue is a large number of solutions computed that will be structurally similar, due to symmetries in the solution space. Second, solvers are usually meant to determine if a CSP has a solution, but a fine tuning of the solver is required to obtain some variety in the proposed solutions.

To overcome these issues, the contributions of this work are to propose a dedicated combinatorial algorithm that is used to enumerate all possible symmetry-free solutions of the CSP defined by the component model, in order to produce initial configurations. This algorithm integrates symmetry elimination patterns which reduce the combinations to be considered. While this algorithm can be used to perform bounded exhaustive testing as in [22], the resulting configuration set can be sampled to select a subset of configurations, that reduce the number of test data to consider. A sampling method, aiming to amplify the variety of configurations, is presented as a second contribution.

Outline. The paper is organized as follows. After a brief overview of the component-based systems under adaptation policies, some basic notions on coverage criteria for their testing are presented in Sect. 2. Section 3 presents the component-based model that is used to represent systems configurations. The configuration generation process, based on bounded exhaustive computation of the possible configurations is presented in Sect. 4. The algorithm is described along with optimisations that aim to reduce the combinatorial explosion, and data selection criteria that make it possible to sample the solutions to a small but significant subset. Section 5 reports on experimental results w.r.t. the research questions. Related works are presented in Sect. 6 before concluding in Sect. 7.

2 Background

On Component-Based Systems Under Adaptation Policies. In this paper, only the basic and generic concepts of component-based systems are considered to allow their application to various hierarchical models: components as entities of several types, required and provided interfaces as interaction points between components, bindings to link component interfaces. Components are either primitive components providing data or services, or composite compound components delegating their interfaces. Components can have some attributes used as configuration parameters. This section presents a running example of such a system, whereas Sect. 3 provides the reader with needed formal notions.

Example 1. Let us start with an example of a Vehicle Platooning Application (VPA for short) inspired from [4]. This complex system is composed of vehicles which are either in solo mode, or organized in some platoons, as displayed in Fig. 2. This figure also provides a component-based architecture corresponding

to the displayed VPA situation. In VPA, each platoon is led by a leader vehicle. Any vehicle in solo mode can ask to join a platoon or decide to create a new platoon with another vehicle in solo mode. Each vehicle in a platoon can ask to quit it either because the vehicle reached his destination, or because it needs to refill its energy. The platoon leader may change either because another vehicle has more autonomy or a further destination. Some external events happen in the system environment, e.g., a new vehicle can arrive on the road, or a driver may decide to quit the platoon on his way to a new destination. These changes on system’s architecture level are considered as dynamic reconfigurations.

For dynamic reconfigurations to occur only in suitable circumstance, adaptation rules indicate, for a given set of configurations, which reconfiguration operations can be triggered, with a utility level associated. Following [6], they are of the form **when** *b* **if** *g* **then utility of** *ope* **is** *f*. As introduced in [18], reconfiguration operations in adaptation policies are guarded by temporal logic properties that may either make use of propositional formulae over configurations, or involve sequences of events and/or reconfigurations.

when after *Join normal* until *Quit normal* **and** *VehicleId.battery < 33*
if *state = leader* **then utility of** *PassRelay* **is** *high*

when after *Join normal* until *Quit normal* **and** *VehicleId.battery > Leader.battery*
if *state = platooned* **then utility of** *GetRelay* **is** *medium*

Example 2. Let us consider 2 adaptation rules involving the *PassRelay* and *GetRelay* reconfigurations. Intuitively, the above rules apply to all vehicles and are used to determine when it is possible to have a relay between the leader and another vehicle of the platoon. In the first case, the *PassRelay* reconfiguration of *high* utility can be triggered when the leader has not enough autonomy. In the

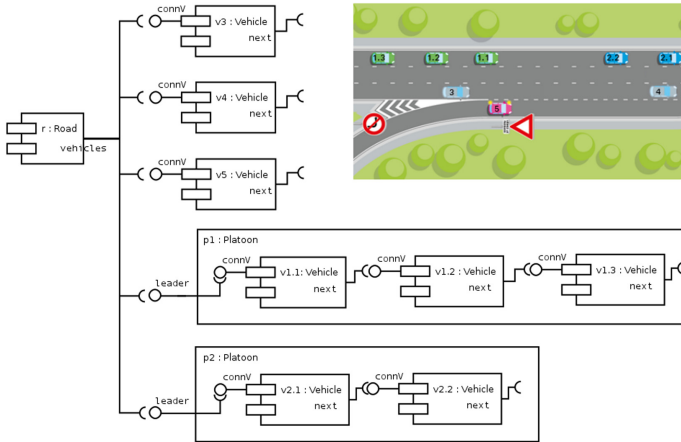


Fig. 2. Component architecture for the considered VPA (top-right side)

second case, the *GetRelay* reconfiguration of medium utility may trigger when the autonomy of a vehicle is greater than the autonomy of the leader.

Notice that a reconfiguration is suggested with a utility value (e.g. from $Ft = \{ \text{high, medium, low} \}$). For the formal definition, the reader can refer to [7].

On Coverage Criteria for Adaptation Policies. In a previous work [7], we have proposed a test generation technique which aims to generate sequences of external events, from usage models of system's components, in order to exercise the reconfiguration rules described in the adaptation policy. The dedicated coverage criteria have been designed for adaptation rules with temporal patterns by exploiting coverage criteria for temporal patterns described in [23].

These coverage criteria can be used as a means to handle the input data, to evaluate a test suite, by measuring how much of the considered artifacts—e.g., temporal properties and adaptation rules with temporal properties—the test suite covers, and to decide when to stop testing. In [23], a temporal property is considered as covered by a test suite TS if each transition of the property test automaton is covered by at least one test case tc from TS . Having the same temporal patterns allows us to consider coverage criteria for adaptation rules and thus for adaptation policies. In [7], the adaptation rule is covered by a test case tc if the rule is eligible—there is a configuration that tc reaches, where b scope and g guard predicates are evaluated to true,—and ope is actually triggered from such a configuration. Coverage criteria for policies are obtained by lifting this notion to sets of rules.

As a consequence, the introduced coverage criteria for adaptation rules allow the user to evaluate if (i) the triggered reconfigurations were among eligible ones, in order to detect undesirable reconfigurations, and (ii) a generated test suite execution has triggered all the reconfiguration rules that were described in the adaptation policy, so as to detect specified but never triggered reconfigurations, even for long test cases generated.

3 Component-Based Model

This section gives means for specifying component-based systems. Their architectural model is defined as a triplet $\langle Elem, Rel, Inst \rangle$, where $Elem$ is a set of the *component elements*, Rel describes the *architectural relationships* between these elements, and $Inst$ is an instantiation of $Elem$ and Rel in terms of actual components and relations.

Components. Components are entities that can be assembled to create an application. As usual, interfaces are used for interactions between components. A provided interface is an interface that the component realizes, whereas a required interface is an interface that the component needs to be able to run. Composite components may delegate their interfaces to inner components. Formally, $Elem = \{ CTypes, IProvided, IRequired, Params, ITypes, PTypes, Contings \}$, where $CTypes$ is the non-empty set of components types, $IProvided$ (resp. $IRequired$) is the set of interfaces, which are provided (resp. required) by

the components, $Params$ is the set of components' parameters, $ITypes$ (resp. $PTypes$) is a finite set of the interfaces (resp. parameters) types, $Contings$ is the set of contingencies that represent the cardinality of required interfaces (single or multiple connections, optional or mandatory).

Example 3 (Components of the VPA example). The architectural elements of the VPA configuration depicted in Fig. 2 are as follows:

$$\begin{aligned}
 CTypes &= \{Road, Platoon, Vehicle\} \\
 IProvided &= \{connV, leader\} \\
 IRequired &= \{vehicles, next\} \\
 Parameters &= \{battery, position, speed, goal\} \\
 ITypes &= \{VInfo\} \\
 PTypes &= \{int, float\} \\
 Contings &= \{singleopt, singlemandatory, multiopt, multimandatory\}
 \end{aligned}$$

Relationships. The architectural relationships among components are defined by a tuple $Rel = \{IPType, IRTType, Provider, Requirer, Contingency, ParamType, Definer, ParentTypes, DelegProv, DelegReq\}$ in which $IPType$ (resp. $IRTType$) is a total function that maps a provided interface in $IProvided$ (resp. a required interface in $IRequired$) to its type in $ITypes$, $Provider$ (resp. $Requirer$) is a total function that maps a provided interface in $IProvided$ (resp. a required interface in $IRequired$) to its component type in $CTypes$. $Contingency$ associates each required interface in $IRequired$ with its contingency. $ParamType$ is a total function that associates with each parameter in $Params$ its type in $PTypes$, $Definer$ is a total function to define the component type in $CTypes$ for each parameter, $ParentTypes$ associates each component type with the component types of its parent components¹, and $DelegProv$ (resp. $DelegReq$) describes pairs of provided interfaces (resp. required interfaces) that are linked by a delegation from a parent component to one of its subcomponents.

Example 4 (3 relationships of the VPA example). For the VPA component model, one has $Provider = \{connV \mapsto Vehicle, leader \mapsto Platoon\}$, $Requirer = \{vehicles \mapsto Road, next \mapsto Vehicle\}$, and $DelegProv = \{connV \mapsto leader\}$.

Instantiation. An instantiation provides the main entities of the component-based system and thus defines its particular configuration, which consists of the components that are present and put together thanks to their relationships. The instantiation is a 6-tuple $Inst = \{Comps, CT, Parents, Binds, DelProv, DelReq, Value\}$ in which $Comps$ is the set of component *instances*; a total function, called CT , associates with each component in $Comps$ its type in $CTypes$; $Parents$ associates with each component in $Comps$ the set of its parent components; $Binds$ is a relation to bind provided and required interfaces of components; $DelProv$ (resp. $DelReq$) describes the delegated interface of a sub-component

¹ Each component type is mapped to a set of component types, as we assume that components can be shared by composite components.

in relation with the delegating interface of the parent component; and *Value* provides the value of each component parameter.

Example 5 (Instantiation of the VPA example). A component can be instantiated several times, as e.g. the components of type *Vehicle*. The configuration in Fig. 2 is given by the following instantiation:

$$\begin{aligned} Comps &= \{v1.1, v1.2, v1.3, v2.1, v2.2, v3, v4, v5, r, p1, p2\} \\ CT &= \{\{v1.1 \mapsto Vehicle, \dots, p1 \mapsto Platoon, \dots, r \mapsto Road\} \\ Parents &= \{v2.1 \mapsto \{p1\}, \dots, v3 \mapsto \emptyset, \dots, p \mapsto \emptyset, r \mapsto \emptyset\} \\ Binds &= \{((v3, connV), (r, vehicles)), \dots, ((v2.2, connV), (v2.1, next)), \dots\} \\ DelProv &= \{((v1.1, connV), (p1, leader)), \dots, ((v2.1, connV), (p2, leader))\} \\ DelReq &= \{\} \\ Value &= \{v1.1 \mapsto \{battery \mapsto 31, position \mapsto 253.3, \dots\}, \dots\} \end{aligned}$$

In addition, following [20], set-theoretical constraints on this architectural model are provided so as to express: (i) the consistent typing of components, (ii) the consistent binding of interfaces, and finally (iii) the consistent parent relationship. For example, only components having a common parent can be bound; mandatory contingencies are fulfilled; a delegated interface of parent component is bound to an appropriate interface of a child component. Finally, some constraints inherent to the considered system are expressed as *system-dependant* invariant properties, like in [20]. They are also needed for the system configurations to be consistent.

We refer to [9, 18] for the definition of components, interfaces, bindings, etc., and their consistent assembly obeying invariants. We call a state or a configuration of a component-based system a set of instantiated above-mentioned architectural elements together with their types and relations to link them.

4 Generation of Initial Configurations

Given the component-based model $\langle Elem, Rel, Inst \rangle$, this section describes a configuration generation algorithm that is used to enumerate all possible symmetry-free solutions of the CSP defined by the component model, in order to produce initial configurations. The aim of this combinatorial algorithm is to build a set of configurations that are correct-by-construction, especially regarding the architectural and consistency constraints that guarantee the correct parenting, delegations and bindings. Nevertheless, the execution of the test cases from all the computed configurations can be a tedious task, especially the setup of the test environment for a given configuration. Hence, some of the solutions can be sampled to produce a reduced set of configurations that are different from each other. These configurations can then be used as initial configurations for the online testing process described in Sect. 1.

4.1 Combinatorial Algorithm

The test generation algorithm is summarized below as Algorithm 1. It takes as an input component model parts *Elem* and *Rel*, and aims to produce all

instantiations $SInst$ up to a given size (expressed as the number of components) that fulfill the description. In order to eliminate irrelevant configurations w.r.t. the system-dependant invariant (notably to restrict possible bindings, e.g. in order to prevent vehicles to be connected to each other outside a platoon), an invariant function can be provided in order to define valid configurations.

Algorithm 1 is parameterized by: the minimal and maximal number of components of each type, the total number of components, a parameter instantiation function, which aims to determine how component parameters are supposed to be valued, and an invariant function which is supposed to provide additional constraints on the configurations, which complement the description of the component model and rule out irrelevant configurations.

```

1: Inputs
2: Elem
3: Rel
4:  $N : int$ 
5: invariant:  $Inst \rightarrow \mathbb{B}$ 
6: genParameters:  $Comp, CT, Definer \rightarrow Values$ 
7: Output
8:  $SInst$  // the set of possible instantiations
9: Begin
10:  $SInst \leftarrow \emptyset$ 
11: for all  $Comp, CT$  from genComponents( $N$ ) do
12:   for all  $Parents$  from genParenting( $Comp, CT$ ) do
13:     if not isFresh( $Parents$ ) then
14:       proceed to the next value of  $Parents$ 
15:     end if
16:     for all  $Delegations$  from genDelegations( $Comp, Parents$ ) do
17:       if not isFresh( $Delegations$ ) then
18:         proceed to the next value of  $Delegations$ 
19:       end if
20:       for all  $Binds$  from genBindings( $Comp, Parents, Delegations$ ) do
21:         if not isFresh( $Binds$ ) then
22:           proceed to the next value of  $Binds$ 
23:         end if
24:          $Values = genParameters(Comp, CT, Definer)$ 
25:          $Inst = \langle Comp, CT, Parents, Delegations, Binds, Values \rangle$ 
26:         if invariant( $Inst$ ) then
27:            $SInst \leftarrow SInst \cup Inst$ 
28:         end if
29:       end for
30:     end for
31:   end for
32: end for
33: End

```

Algorithm 1: Initial configurations generation

The combinatorial algorithm proceeds by successive steps. Each step consists in identifying the possible solutions before considering the valid one, one-by-one to proceed to the next step. Once a given step has explored all the possibilities, the algorithm backtracks to the previous step to consider the next solution before moving onto the next step. The different steps are the following.

Step 1. The algorithm starts (line l.11) by considering all possible partitions of the components according to their types, bound by a maximal cardinality size. This step relies on the *CTypes* description in *Elem*. Each pair from *Comps* \times *CT* (of components and types in *Inst*) is considered for the subsequent step.

Step 2. The second step (l.12) aims to produce, for a given set of instances, a parenting relationship that fulfills the following constraints: (i) each composite component has at least two children, and (ii) no loop may appear in the parenting relationship. At this step, the *isFresh* function (l.13) is used to detect if the solution that is computed has been already encountered modulo permutation in a previous iteration of the current loop, as illustrated by Fig. 3. If so, the solution is not considered for the subsequent step, and the algorithm proceeds the next parenting solution.

Step 3. The third step (l.16) consists in delegating required or provided interfaces of the composite components to one of its children. To save space *Delegations* represents both *DelProv* and *DelReq* described previously. All interfaces of the composites have to be delegated to their inner components. Similarly to the previous step, symmetrical solutions are ruled out, as illustrated by Fig. 4.

Step 4. The fourth step (l.20) consists in computing, based on the current parenting and delegations, a binding of compatible required and provided interfaces, that satisfies architectural constraints (e.g. only components with the same parent can be bound) and contingency constraints (single/multiple, mandatory/optional). Here again, symmetrical solutions, as illustrated by Fig. 5, are not considered.

Step 5. Once the structure of component system is generated, the algorithm eventually computes data values for the component parameters, according to their type, based on a given valuation function that can be user-defined (l.24-25). For both discrete and continuous domains of the considered parameters, this function makes use of the *Beta*-distribution method [12] with parameterized probabilistic distributions allowing the user to vary the density of random draw from these domains, and to automate the process. In the end, the configuration that has been computed is checked against the invariant (l. 26), before being stored (l. 27).

Finally, only valid and consistent configurations up to size N are computed².

² The interested reader can find an implementation of this algorithm at <https://fdadeau.github.io/CSConfigGen/>.

Proposition 1. *Given number N of components' instances to be generated, Algorithm 1 terminates either by providing a set $SInst$ of consistent configurations with up to N components (without those obtained by permutations of architectural elements), or by returning the empty set if none of the configurations satisfies consistency constraints or system-dependant invariant.*

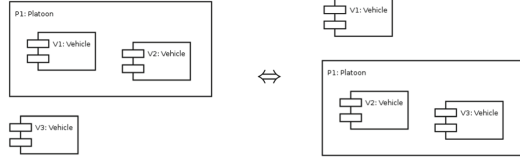


Fig. 3. Symmetries in parenting relationships

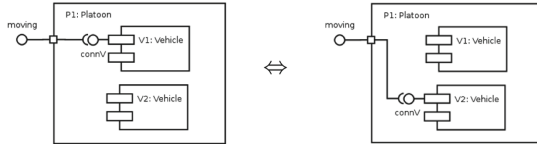


Fig. 4. Symmetries in delegations

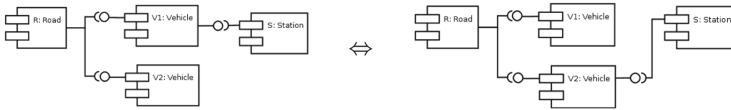


Fig. 5. Symmetries in bindings

4.2 Initial Configuration Sampling

Once initial configurations generated, there is a need to select some of them for testing the system under adaptation policies. Intuitively, while using coverage criteria for handling test inputs, the greater the difference between the configurations, the higher the coverage rate will be.

Comparing Configurations. The difference between the configurations is computed by processing them two by two. This computation can be divided into 2 parts: on the overall architecture, and on particular system-dependant features. First, we compare the overall structure of the configurations by counting the difference Δ_{comps} between the numbers of the components instances (primitive and compound ones), and the difference $\Delta_{hierarchy}$ between the numbers of all the ancestors of the involved components. This way, a better coefficient will be given to a complex configuration with nested composite components, compared with

a flat configuration. Depending of the systems under consideration, the number of bindings may differentiate the configurations as well, hence $\Delta_{bindings}$.

In [1], a negative inverse exponential function is used to limit the complexity score, together with a coefficient to scale it between two values. In the same spirit, we have chosen a logarithmic function to limit our coefficient values. So, in the end, a coefficient on the overall architecture difference is computed by the following formulae:

$$k = \log_{10}(\Delta_{comps}^{\Delta_{hierarchy} + \Delta_{bindings}})$$

Example 6. Let us consider again the configuration in Fig. 2 with 8 vehicles, 2 platoons, 1 road, and 3 bindings (configuration *A*). Let us compare it to configuration *B* composed of 5 solo vehicles on the road. One has $\Delta_{comps} = 5$ for primitive and compound components, $\Delta_{hierarchy} = 5$, and $\Delta_{bindings} = 3$, so $k = \log_{10}(5^{5+3}) = 5.59$.

Second, while testing adaptation rules, the test case generation may be impacted by the values of the component parameters, that are involved in the rules. So, the validation engineer should be able to examine the parameters that are worthy of attention. To compare two configurations with a different number of the components of the same type, for each parameter of interest of the configuration with more instances, the closest values are two by two selected for merging, until the same number of values is obtained. So, given number n of instances of the same type considered, l_1 and l_2 sorted lists of n values of parameter Par , the proportional difference is computed by the following formulae:

$$score_{Par} = \frac{100}{n \times (max_{Par} - min_{Par})} \times \sum_{i=1}^n |l_{1_i} - l_{2_i}|$$

where max_{Par} and min_{Par} represent resp. maximal and minimal values of parameter Par . The scores of all parameters from $Pars \subseteq Params$ that impact the adaptation rule, are aggregated in a final score, where k is the coefficient on the overall architecture difference between two configurations:

$$difScore = k \times \sum_{Par \in Pars} score_{Par}$$

Example 7. Let us consider again configurations *A* and *B* from Example 6, with a focus on *battery* parameter. Let us suppose that the battery level values for the vehicles in *A* are (20, 22, 34, 54, 62, 72, 80, 99). As there are 3 more vehicles in *A*, the 3 pairs of the closest values are: (20, 22), (55, 62) and (72, 80). After the merging step (here by averaging), one has $l_1 = (21, 34, 58, 76, 99)$. For *B*, let us take $l_2 = \{26, 55, 62, 74, 89\}$. Applying $score_{Par}$ to *battery* gives:

$$score_{Bat} = \frac{100}{5 \times (100 - 20)} \times \sum_{i=1}^5 |l_{1_i} - l_{2_i}| = 0.25 \times 42 = 8.4$$

Assuming *distance* parameter score being $score_{distance} = 5.1$, the aggregated difference score is then: $difScore = 5.59 \times (8.4 + 5.1) = 75.46$.

Configuration Sampling. Sampling consists in reducing the set of possible configurations to a subset of size NS , in which the difference scores between the configurations are maximized. Such an optimization problem can be solved by various kind of approaches, such as SAT-solving, linear programming, clustering [13], genetic programming [19], etc. For this work, we choose to implement a greedy algorithm, shown below as Algorithm 2. Based on the set of m generated initial configurations computed, an $m \times m$ score matrix (named *Scores* line 2) is built, where $a_{i,j}$ element represents *diffScore* between configuration i and configuration j ³. NS (line 2) denotes the number of configurations to select, which is the cardinality of *Configs* set

```

1: Inputs
2:   Scores, NS
3: Output
4:   Configs
5: Begin
6:  $i, j \leftarrow \text{selectHighestScore}(\textit{Scores})$ 
7: mark  $a_{i,j}$  as selected
8: Configs.add(i)
9: Configs.add(j)
10: repeat
11:    $j' \leftarrow \text{selectHighestScoreInLine}(i)$ 
12:   mark  $a_{i,j'}$  as selected
13:   Configs.add(j')
14:    $i' \leftarrow \text{selectHighestScoreInColumn}(j)$ 
15:   mark  $a_{i',j}$  as selected
16:   Configs.add(i')
17:    $i \leftarrow i'$ 
18:    $j \leftarrow j'$ 
19: until Configs.size() <  $NS$ 
20: return Configs
21: End

```

Algorithm 2: Initial config. sampling

of indexes of selected configurations (line 6). The algorithm starts by selecting the biggest score in *Scores* matrix with function *selectHighestScore* (line 6), and marks the corresponding element as selected (line 7). *Configs* set is then updated (lines 8, 9). Here *selectHighestScore(Scores)* function browses the matrix given parameters, and returns the indexes corresponding to the biggest difference score between i -th and j -th configurations. Then, in the corresponding row i and column j the biggest remaining scores are chosen (lines 11 and 14), and the corresponding configurations indexes are added to *Configs* (lines 13 and 16). Function *selectHighestScoreInLine(i)* (resp. *selectHighestScoreInColumn(j)*) browses the i -th row (resp. the j -th column) of *Scores* matrix, and returns the index of the column (j') (resp. row i') corresponding to their respective biggest score. In order to prepare the next iteration step, the indexes are updated (lines 17, 18). The steps in lines 11 to 18 are repeated, until *Configs* size reaches NS .

By construction, only configurations with big difference scores are selected.

Proposition 2. *Given NS , number of configurations to select from $SInst$ set of size m , Algorithm 2 terminates by providing *Configs* set of size $NS \leq m$ of configuration indexes from $SInst$ that have the most significant difference scores.*

³ In this matrix $a_{i,i} = 0$ and $a_{i,j} = a_{j,i}$. The complexity of the computation is quadratic in the number m of configurations.

4.3 Integration into the Online Test Generation Process

The online testing process relies on the usage models, one per component type, which are probabilistic automata. They capture the behavior of components and determine, for a given state, which external events can be sent to the component, at a given rate. As an example, Figure 6 represents the usage model of Vehicle type components. In this figure, edges with solid lines represent external events that may be used for stimulating (i.e., testing) the component, whereas dotted lines represent internal events that may occur and change the state of the automaton. Edges may also be labelled by δ , which represents a quiescence, meaning that no event will be sent to the system. Finally, the number in parentheses represents the probability for the considered event to be selected.

The usage models are specific to each component type, and each state of the automaton represents a given configuration for the component. As a consequence, we assume that there is, for each component type, a function to determine the initial state of usage model for a component of this type. We denote $init_{CType}$ this function.

Example 8. Assuming that a component v of type Vehicle appears in the instantiation $Inst$ that is generated by the process described in Sect. 4.1 and selected by the process described in Sect. 4.2, this component’s automaton initial state will be given by the following function:

```
function  $init\_Vehicle(v, Inst)$ 
  if  $((v, connV) \notin Inst.Binds)$  return 0
  else if  $(Inst.Parents(v) \neq \emptyset)$  return 1
  else return 2
```

5 Experimentation

This section describes experiments to assess the testing approach described in Sect. 1 and displayed in Fig. 1 while using Algorithms 1 and 2 for initial configuration automatic generation and sampling. The goal of the experiments is to answer the following research questions.

[RQ1] *To what extent the use of different initial configurations improves the generated tests? (shorter? improve coverage? find more faults?)*

[RQ2] *To what extent the symmetry-breaking in Algorithm 1 reduces the number of generated configurations?*

On the Experiments. To experiment, a simulator of the VPA example has been developed as a Java program (almost 6000 lines of code). It can be modified at will, e.g. to set up initial configurations and sequences of events. It is also possible for the validation engineer to modify the implementation of the adaptation policies that guide system’s reconfigurations. Actually, the implementation may depend on the reconfigurations utilities and on strategies for handling priorities of the reconfigurations with the same utility level. The validation of implementation choice has been described in [8].

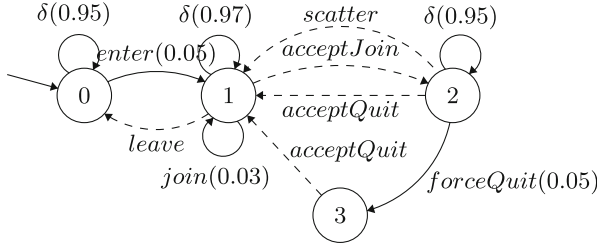


Fig. 6. Usage model for the Vehicle components

For the VPA system under test, 8 adaptation rules have been designed, that integrate 5 temporal properties of interest.

Example 9. In addition to 2 adaptation rules for vehicles from Example 2 involving *battery* parameter, we consider the following rule with *distance* parameter: **when** after *Join normal* until *Quit normal* **and** *VehicleId.distance < 10* **if** *state != leader* **then utility of** *QuitPlatoon* **is** *high*

On the Experimental Protocol. Let us now describe the experimental protocol. Once the set of initial configurations *Inst* of cardinality 1200 generated by applying Algorithm 1, matrix *Scores* of difference scores is computed. Afterwards, 10 closest configurations with small difference scores and 10 farthest ones with big difference scores are selected by applying twice Algorithm 2 to *Score* matrix.

As the test generator performs Markovian walk over components usage models, the experiment is replayed 170 times for each set of configurations, one by one ($2 \times 10 \times 170$), to provide a confidence in the experimental results. This allows observing produced traces with actually triggered reconfigurations in order to compute the coverage criteria rate [7] as recalled in Sect. 2.

For the VPA example simulation, as displayed in Fig. 1, running an experiment consists then in starting from a selected initial configuration and letting the test generator deal with usage models of components to send the events at a given rate to the system under adaptation policies. During 3000-step experiments, the reconfigurations occur (with traces produced) and make system's architecture evolve.

On the Results. The coverage rate is separately aggregated for the properties and for the adaptation rules by applying coverage criteria described in [7] and reminded in Sect. 2. Given the set of initial configurations, for each experiment the coverage rate for the rules is the ratio of the number of adaptation rules, that are covered by at least one test case starting from a configuration from this set, to the number of rules under consideration.

Table 1 below reports on the experimental results⁴, where the lines correspond to the coverage rate reached for the properties and rules, depending on the configuration set chosen with either small *difScore* values, or big *difScore* values. The columns represent the running experiment number (from 1 to 170), with an extract of 9 experiments below. For example, for the run in column $n+1$, the first line (Small dif. score) indicates 94% of coverage for properties and 75% for rules, where as the second line (Big dif. score) indicates 100% coverage for properties and 75% for adaptation rules. The Av. column indicates the average coverage rate of a sub-line. For example, in the first line (Small dif. score) the first sub-line indicates 86.5% properties coverage on average for 170 performed experiments. Also, for the 8 adaptation rules, 0% coverage rate indicates that no rule has been triggered during the 3000-step experiment with about 300 external events sent to the SUT⁵, whereas 100% coverage says that all the adaptation rules have been triggered. The column M.freq. indicates the most often seen rate value over 170 experiments performed for each set of configurations. So, 75 indicates that 75% is the most frequent coverage.

Let us note that the most frequent 75%-rate for adaptation rules is due to the rules, whose *QuitDistance* reconfiguration is not triggered because of *distance* parameter involved. For these rules, 3000-step experiments are not long enough for decreasing *distance* values, and the scope and guard predicates of the concerned rules remain false.

Table 1. Extract of experimental results and coverage rates

Run number		23	24	25	26	27	28	29	30	31	Av.	M.fr.
Small dif.	Prop.Cov.(%) ...	56	94	17	94	56	61	94	100	94	...	86.5 94
score	Rule Cov.(%) ...	0	75	0	75	0	13	75	75	75	...	58.5 75
Big dif.	Prop.Cov.(%) ...	100	100	100	100	100	100	94	100	100	...	98.1 100
score	Rule Cov.(%) ...	94	75	75	75	75	100	75	75	75	...	80.2 75

On the Symmetry Elimination. To address **RQ2**, we ran an experiment, which consists in counting the number of configurations that are generated with or without the symmetry detections that we have considered.

For the VPA example we designed 5 setups that differ in the minimal and maximal number of components of each type that are generated. The invariant specifies that vehicles that are not in a platoon are not connected together. The setups are as follows: Setup#1: 1 Road, 1 to 5 Vehicles, 0 to 2 Platoons, 0 to 1 Station. Setup#2: 1 Road, 5 Vehicles, 0 to 2 Platoons, 0 to 1 Station. Setup#3: 1 Road, 6 Vehicles, 0 to 2 Platoons, 0 to 1 Station. Setup#4: 1 Road, 1 to 5 Vehicles, 1 Platoon, 0 to 1 Station. Setup#5: 1 Road, 1 to 5 Vehicles, 0 to 2 Platoons, 1 Station.

⁴ More results are in Table at <https://fdadeau.github.io/CSCConfigGen/table.html>.

⁵ Let us note that for each experiment, on the given clock tick, on average 10% of steps correspond to the events from the usage models sent to the SUT (cf. (4) in Fig. 1), whereas δ occurs for the remaining 90% of steps.

By turning on or off (denoted with a line over the corresponding symbol) some of the symmetry eliminations (parenting P , delegations D , bindings B) and invariant filtering (I), we obtain the results shown in Fig. 7. Setup #1, #3 and #5 took about 25 s on a standard laptop (Dual-core i5 1.6 GHz with 8Go RAM) to generate 21.000–23.000 configurations.

Setup	#1	#2	#3	#4	#5
P, D, B, I	62	26	39	30	44
\overline{P}, D, B, I	181	93	166	54	149
$\overline{\overline{P}}, D, B, I$	325	244	1098	158	222
$\overline{\overline{\overline{P}}}, D, B, I$	640	482	2398	378	491
$\overline{\overline{\overline{\overline{P}}}}, D, B, I$	1083	897	5270	410	700
$\overline{\overline{\overline{\overline{\overline{P}}}}}, D, B, I$	2249	1953	17495	1294	1466
$\overline{\overline{\overline{\overline{\overline{\overline{P}}}}}}, D, B, I$	22971	21213	337625	4174	21218

Fig. 7. Number of configurations for each setup

Due to the exponential blow up of the unfolding, it takes about 20 min to generate the 337.625 configurations of Setup #3, which was reduced to 16 min when enabling symmetry reductions.

All symmetry reductions are clearly relevant in order to master the combinatorial explosion, showing that only a tractable number of configurations is generated even for highly combinatorial setups. On this case study, the experimentation also shows that a large set of irrelevant configurations can be generated, based only on the description of the component model (without considering the invariant).

Notice that these symmetrical configurations have 0 for the difference score, and thus, only one of them will at most be kept by the selection process. As a consequence, removing them at the soonest prevents useless computations from being performed.

The obtained experimental results allow assessing the use of automatically generated initial configurations and their sampling. Indeed, they show that the set of the generated configurations with significant difference scores gives better coverage rates, thus answering our research questions.

6 Related Work

System’s configurations are required for online testing approaches to work. In our approach, these configurations are automatically generated from a component model by using boundary testing [3,21,24] to generate system’s values. In [21], the authors present a test case generation based on boundary goals derived from a formal model, with a feedback to refine boundary criteria if the boundary goals are not reachable. In our approach, as in [14], the boundaries of the parameters are defined in the model; in this respect our contribution consists in generating a wide diversity of combinations.

Our symmetry filtering approach for generating initial configurations is close to the TACO tool [11], which applies SAT-based techniques instrumented with a symmetry-breaking predicate to JML-annotated sequential Java programs in order to eliminate some isomorphic models. Our approach goes further, as we also consider hierarchical objects.

In the field of software product lines (SPLs), the configuration spaces are often determined by features and constraints over them, modeled as feature

models (FMs). A feature oriented testing (FOT) described in [16] applies FMs to test-case designs for black-box testing. In this approach, SAT-based automated test-suite generation and correctness checking of test-case designs are performed. In [15] the authors present a comparative study of combinatorial testing (CT) and random testing (RT) algorithms for testing SPLs. On the chosen benchmarks, this study shows that the diversity of configurations sampled by CT is 2 to 3 times higher than those sampled by RT.

As reported in [17], in fuzz testing, the performance substantially varies depending on input configuration files, or seeds, used to start fuzzing with. However, most papers (among 32) have treated their choice casually, apparently assuming that any seed would work equally well, without providing particulars. Our experimentations with initial configurations also show that testing process heavily depends on them.

7 Conclusion and Future Works

In this paper, we have presented an approach to automatically generate initial configurations for testing component-based systems. The presented algorithm allows generating structured data composed of architectural elements and relationships to link them, while satisfying general consistency constraints expressed by invariants. System-dependant instantiation of component parameters is integrated at appropriate algorithm steps, in order to generate meaningful inputs for testing. The provided experimental results on a simulation of platoons of autonomous vehicles show that this approach allows us to improve the capability of fulfilling coverage criteria by using different initial configurations. Thus, the present work usefully extends the approach for testing component-based systems in [7]. This approach is generic for any component-based framework, and can be extended to adapt to component models, such as BIP [2].

One of the future work directions consists in providing the user with a refinement method in order to enlarge or reduce defined boundaries. Also, we intend to improve detection of dubious reconfigurations, and to provide the user with the means to validate that an adaptation policy, that is correctly implemented, fulfills extra-functional properties, such as optimized resource-consumption, etc.

References


1. Alkan, B., Harrison, R.: A virtual engineering based approach to verify structural complexity of component-based automation systems in early design phase. *J. Manuf. Syst.* **53**, 18–31 (2019)
2. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2006*, Washington, DC, USA, pp. 3–12. IEEE Computer Society (2006)
3. Beizer, B., Wiley, J.: *Black box testing: techniques for functional testing of software and systems*. *IEEE Softw.* **13**(5), 98 (1996). <https://doi.org/10.1109/MS.1996.536464>

4. Bergenhem, C.: Approaches for facilities layer protocols for platooning. In: IEEE 18th International Conference on Intelligent Transportation Systems, ITSC 2015, pp. 1989–1994. IEEE (2015). <https://doi.org/10.1109/ITSC.2015.322>
5. Bouquet, F., Legeard, B., Peureux, F.: CLPS-B - A constraint solver to animate a B specification. *Int. J. Softw. Tools Technol. Transf.* **6**(2), 143–157 (2004). <https://doi.org/10.1007/s10009-003-0123-8>
6. Chauvel, F., Barais, O., Borne, I., Jézéquel, J.: Composition of qualitative adaptation policies. In: 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), pp. 455–458. IEEE Computer Society (2008)
7. Dadeau, F., Gros, J.P., Kouchnarenko, O.: Testing adaptation policies for software components. *Softw. Qual. J.* **28**(3), 1347–1378 (2020). <https://doi.org/10.1007/s11219-019-09487-w>
8. Dadeau, F., Gros, J.P., Kouchnarenko, O.: Online testing of dynamic reconfigurations w.r.t. adaptation policies. *Model. Anal. Inf. Syst.* **28**(1), 52–73 (2021). <https://doi.org/10.18255/1818-1015-2021-1-52-73>. (in Russian)
9. Dormoy, J., Kouchnarenko, O., Lanoix, A.: Using temporal logic for dynamic reconfigurations of components. In: Barbosa, L.S., Lumpe, M. (eds.) FACS 2010. LNCS, vol. 6921, pp. 200–217. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27269-1_12
10. El Ballouli, R., Bensalem, S., Bozga, M., Sifakis, J.: Programming dynamic reconfigurable systems. *Int. J. Softw. Tools Technol. Transf.* (2021). <https://doi.org/10.1007/s10009-020-00596-7>
11. Galeotti, J.P., Rosner, N., López Pombo, C.G., Frias, M.F.: TACO: efficient sat-based bounded verification using symmetry breaking and tight bounds. *IEEE Trans. Softw. Eng.* **39**(9), 1283–1307 (2013). <https://doi.org/10.1109/TSE.2013.15>
12. Gupta, A., Nadarajah, S.: *Handbook of Beta Distribution and Its Applications*. CRC Press (2004)
13. Hartigan, J.A., Wong, M.A.: A k-means clustering algorithm. *JSTOR Appl. Stat.* **28**(1), 100–108 (1979)
14. Hussain, A., Tiwari, S., Suryadevara, J., Enouï, E.: From modeling to test case generation in the industrial embedded system domain. In: Mazzara, M., Ober, I., Salaiün, G. (eds.) STAF 2018. LNCS, vol. 11176, pp. 499–505. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04771-9_35
15. Jin, H., Kitamura, T., Choi, E.-H., Tsuchiya, T.: A comparative study on combinatorial and random testing for highly configurable systems. In: Casola, V., De Benedictis, A., Rak, M. (eds.) ICTSS 2020. LNCS, vol. 12543, pp. 302–309. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64881-7_20
16. Kitamura, T., Do, N.T.B., Ohsaki, H., Fang, L., Yatabe, S.: Test-case design by feature trees. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012. LNCS, vol. 7609, pp. 458–473. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34026-0_34
17. Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 2123–2138 (2018)
18. Kouchnarenko, O., Weber, J.-F.: Adapting component-based systems at runtime via policies with temporal patterns. In: Fiadeiro, J.L., Liu, Z., Xue, J. (eds.) FACS 2013. LNCS, vol. 8348, pp. 234–253. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07602-7_15
19. Koza, J.R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge (1992)

20. Lanoix, A., Dormoy, J., Kouchnarenko, O.: Combining proof and model-checking to validate reconfigurable architectures. *Electron. Notes Theor. Comput. Sci.* **279**(2), 43–57 (2011). <https://doi.org/10.1016/j.entcs.2011.11.011>
21. Legeard, B., Peureux, F., Utting, M.: Automated boundary testing from Z and B. In: Eriksson, L.-H., Lindsay, P.A. (eds.) *FME 2002*. LNCS, vol. 2391, pp. 21–40. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45614-7_2
22. Sullivan, K., Yang, J., Coppit, D., Khurshid, S., Jackson, D.: Software assurance by bounded exhaustive testing. In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, pp. 133–142. Association for Computing Machinery, New York (2004)
23. Taha, S., Julliand, J., Dadeau, F., Cabrera Castillos, K., Kanso, B.: A compositional automata-based semantics and preserving transformation rules for testing property patterns. *Formal Aspects Comput.* **27**(4), 641–664 (2015). <https://doi.org/10.1007/s00165-014-0328-5>
24. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.* **22**(5), 297–312 (2012). <https://doi.org/10.1002/stvr.456>
25. Walton, G.H., Poore, J.H., Trammell, C.J.: Statistical testing of software based on a usage model. *Softw. Pract. Exp.* **25**(1), 97–108 (1995)



Monitoring Distributed Component-Based Systems

Yliès Falcone¹ , Hosein Nazarpour², Saddek Bensalem²,
and Marius Bozga²

¹ Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France
yliès.falcone@univ-grenoble-alpes.fr

² Univ. Grenoble Alpes, CNRS, Grenoble INP, Verimag, 38000 Grenoble, France
{hosein.nazarpour,saddek.bensalem,marius.bozga}@univ-grenoble-alpes.fr

Abstract. We monitor asynchronous distributed component-based systems with multi-party interactions. We consider independent components whose interactions are managed by several distributed schedulers. In this context, neither a global state nor the total ordering of the executions of the system is available at runtime. We instrument the system to retrieve local events from the local traces of the schedulers. Local events are sent to a global observer which reconstructs on-the-fly the set of global traces that are compatible with the local traces, in a concurrency-preserving fashion. The set of compatible global traces is represented in the form of an original lattice over partial states, such that each path of the lattice corresponds to a possible execution of the system.

1 Introduction

Component-based design consists in constructing complex systems using a set of predefined components. Each component is an atomic entity with some actions and interfaces. Components communicate and interact with each other through their interfaces. The behavior of a component-based system with multiparty interactions (CBS) is defined according to the behavior of each component as well as their interactions. Each interaction is a set of simultaneously executed actions of the existing components [9]. In the distributed setting, for efficiency reasons, the execution of the interactions is distributed among several independent schedulers. Schedulers and components are interconnected (e.g., networked physical locations) and work together as a whole unit to meet some requirements. The execution of a multi-party interaction is then achieved by sending/receiving messages between the schedulers and the components [3].

Verification techniques can ensure the correctness of a distributed CBS. Runtime Verification (cf. [1, 18, 30]) consists in verifying the executions of the system against the desired properties. We consider properties referring to the global states of the system which can not be projected nor checked on individual components. In the following, we point out the problems that one encounters when monitoring distributed CBSs. We use neither a global clock nor a shared memory. This makes the execution of the system more dynamic and parallel by avoiding

synchronization to take global snapshots, which would go against the distribution of the verified system. However, it complicates the monitoring problem because no component of the system can be aware of the global trace. Since the execution of interactions is based on sending/receiving messages, communications are asynchronous and delays in the reception of messages are inevitable. Moreover, the absence of ordering between the execution of the interactions in different schedulers makes the actual execution trace not observable. Our goal is to allow for the verification of distributed CBSs by formally instrumenting them to observe their global behavior while preserving their performance and behavior.

Our main contribution is an approach for the monitoring of distributed CBSs w.r.t. specifications referring to the global states of the system. First, we define a *monitoring hypothesis* that permits to rely on an abstract semantic model of distributed CBSs that encompasses a variety of distributed (component-based) systems. Our model only relies on the semantics of CBS, given in terms of Labeled Transition Systems (LTSs), thus it is not bound to any CBS framework. In a distributed CBS, due to the parallel executions in different schedulers (i) events (i.e., actions changing the state of the system) are not totally ordered, and (ii) the actual execution trace of a distributed system can not be obtained. Although each scheduler is aware of its local events, to evaluate the global behavior, it is necessary to find a set of possible ordering of the events of all schedulers, that is, the set of compatible execution traces. In our setting, schedulers do not communicate together but only communicate with their own associated components. Indeed, what makes the actions of different schedulers to be causally related is only the shared components, which are involved in several multi-party interactions managed by different schedulers. In other words, the executions of two actions managed by two schedulers and involving a shared component are definitely causally related, because each execution requires the termination of the other execution in order to release the shared component. To account for existing causalities among events, we (i) employ *vector clocks* to define the ordering of events, (ii) compose each scheduler with a *controller* to compute the correct vector clock of each generated event, (iii) compose each shared component with a controller to resolve the causality, and (iv) introduce a centralized algorithm that executes on a *global observer* to reconstruct a set of compatible execution traces that could possibly happen in the system with respect to the received events. We represent the set of compatible traces using a computation lattice tailored for CBSs. Such a computation lattice consists of a set of partially connected nodes. Created nodes are partial states and become global states during monitoring. Any path of the lattice projected on a scheduler represents the corresponding local partial trace according to that scheduler (*soundness*). All possible global traces are recorded (*completeness*).

An extended version of this paper with more details and proofs is available in [29].

2 Preliminaries and Notations

Sequences. For a finite set E , a sequence s containing elements of E is formally defined by a total function $s : I \rightarrow E$ where I is either the integer interval $[0..n]$

for some $n \in \mathbb{N}$, or \mathbb{N} itself (the set of natural numbers). Given a set of elements E , $e_1 \cdot e_2 \cdots e_n$ is a sequence or a list of length n over E , where $\forall i \in [1..n]. e_i \in E$. The empty sequence is noted ϵ or $[\]$, depending on the context. The set of (finite) sequences over E is noted E^* . E^+ is defined as $E^* \setminus \{\epsilon\}$. The length of a sequence s is noted $\text{length}(s)$. We define $s(i)$ as the i^{th} element of s and $s(i \cdots j)$ as the factor of s from the i^{th} to the j^{th} element; and $s(i \cdots j) = \epsilon$, if $i > j$. We define function $\text{last} : E^+ \rightarrow E$ as $\text{last}(e) = s(\text{length}(s))$. For an infinite sequence $s = e_1 \cdot e_2 \cdot e_3 \cdots$, we define $s(i \cdots) = e_i \cdot e_{i+1} \cdots$ as the suffix of s from index i onwards. An n -tuple is an ordered list of n elements, where $n \in \mathbb{N}$. The i^{th} element of tuple t is denoted by $t[i]$.

Labeled Transition Systems (LTS). Labeled Transition Systems (LTSs) are used to define the semantics of CBSs. An LTS is a 3-tuple (State, Lab, Trans) where State is a non-empty set of states, Lab is a set of labels, and $\text{Trans} \subseteq \text{State} \times \text{Lab} \times \text{State}$ is the transition relation. A transition $(q, a, q') \in \text{Trans}$ means that the LTS can move from state q to state q' by consuming label a ; we say that a is *enabled* in q . We abbreviate $(q, a, q') \in \text{Trans}$ by $q \xrightarrow{a}_{\text{Trans}} q'$ or by $q \xrightarrow{a} q'$ when clear from context. Moreover, relation Trans is extended to its reflexive and transitive closure in the usual way and we allow for regular expressions over Lab to label moves between states: if expr is a regular expression over Lab (i.e., expr denotes a subset of Lab^*), $q \xrightarrow{\text{expr}} q'$ means that there exists one sequence of labels in Lab matching expr such that the system can move from q to q' .

Vector Clock. Mattern and Fidge's vector clocks [20, 27] are a more powerful extension of Lamport's scalar logical clocks [23], i.e., strongly consistent with the ordering of events. In a distributed system with a set of schedulers $\{S_1, \dots, S_m\}$, $VC = \{(c_1, \dots, c_m) \mid \forall j \in [1..m]. c_j \in \mathbb{N}\}$ is the set of vector clocks, such that vector clock $vc \in VC$ is a tuple of m scalar (initially zero) values c_1, \dots, c_m locally stored in each scheduler $S_j \in \{S_1, \dots, S_m\}$ where $\forall k \in [1..m]. vc[k] = c_k$ holds the latest (scalar) clock value scheduler S_j knows about scheduler $S_k \in \{S_1, \dots, S_m\}$. A unique vector clock is associated each event in the system ([27], Sect. 7). For two vector clocks vc_1 and vc_2 , $\max(vc_1, vc_2)$ is a vector clock vc_3 such that $\forall k \in [1..m]. vc_3[k] = \max(vc_1[k], vc_2[k])$. Moreover two vector clocks can be compared together such that $vc_1 < vc_2 \iff \forall k \in [1..m]. vc_1[k] \leq vc_2[k] \wedge \exists z \in [1..m]. vc_1[z] < vc_2[z]$.

Happened-Before Relation [23]. Relation \succ on the set of system events is the smallest relation satisfying the following three conditions: (1) If a and b are events in the same scheduler, and a comes before b , then $a \succ b$. (2) If a is the sending of a message by one scheduler and b is the reception of the same message by another scheduler, then $a \succ b$. (3) If $a \succ b$ and $b \succ c$ then $a \succ c$. Two distinct events a and b are said to be concurrent if $a \not\succ b$ and $b \not\succ a$. Vector clocks are strongly consistent with happened-before relation. That is, for two events a and b with associated vector clocks vc_a and vc_b respectively, $vc_a < vc_b \iff a \succ b$.

Computation Lattice [27]. A computation lattice is represented as a directed graph with m (i.e., number of schedulers executed in distributed manner) axes. Each axis is dedicated to the state evolution of a scheduler. A computation lattice expresses all the possible traces. A computation lattice \mathcal{L} is a pair (N, \succ) , where N is the set of nodes (i.e., global states) and \succ is the happened-before relation among the nodes.

3 Distributed CBS

We describe our assumptions on CBSs by providing them with a general semantics. The exact model and the system behavior are unknown. The architecture, the behaviors of the components and the schedulers, and the association between schedulers and components can be obtained by several techniques such as the ones in [7, 10]. Our monitoring framework is independent from the technique used to obtain the system and its implementation. Inspiring from conformance-testing theory [32], we refer to this as the *monitoring hypothesis*.

3.1 Semantics

The system is composed of *components* in a set $\mathbf{B} = \{B_1, \dots, B_{|\mathbf{B}|}\}$ and *schedulers* in a set $\mathbf{S} = \{S_1, \dots, S_{|\mathbf{S}|}\}$. Each component B_i is endowed with a set of actions Act_i . Joint actions, aka multi-party interactions, involve the execution of actions on several components. An interaction is a non-empty subset of $\bigcup_{i=1}^{|\mathbf{B}|} Act_i$ and we denote by Int the set of interactions in the system. At most one action of each component is involved in an interaction: $\forall a \in Int. |a \cap Act_i| \leq 1$. In addition, each component B_i has internal actions modeled as a unique action β_i . Schedulers coordinate the execution of interactions and ensure that each multi-party interaction is jointly executed (Definition 2).

We assume some functions from the system architecture.

- Function $inv : Int \rightarrow 2^{\mathbf{B}} \setminus \{\emptyset\}$ indicates the components involved in an interaction. Moreover, we extend function inv to internal actions by setting $inv(\beta_i) = i$, for any $\beta_i \in \{\beta_1, \dots, \beta_{|\mathbf{B}|}\}$. Interaction $a \in Int$ is a joint action if and only if $|inv(a)| \geq 2$.
- Function $mng : Int \rightarrow \mathbf{S}$ indicates the scheduler managing an interaction: for an interaction $a \in Int$ $mng(a) = S_j$ if a is managed by scheduler S_j .
- Function $scp : \mathbf{S} \rightarrow 2^{\mathbf{B}} \setminus \{\emptyset\}$ indicates the set of components in the scope of a scheduler s.t. $\forall j \in [1..|\mathbf{S}|]. scp(S_j) = \bigcup_{a' \in \{a \in Int \mid mng(a) = S_j\}} inv(a')$.

We describe the behavior of components, schedulers, and their composition.

Definition 1 (Behavior of a component). *The behavior of a component B is an LTS $(Q_B, Act_B \cup \{\beta_B\}, \rightarrow_B)$ s.t.:*

- Q_B is the set of states which has a partition $\{Q_B^r, Q_B^b\}$, where Q_B^r (resp. Q_B^b) is the so-called set of ready (resp. busy) states,

- Act_B is the set of actions, and β_B is the internal action,
- $\rightarrow_B \subseteq (Q_B^r \times Act_B \times Q_B^b) \cup (Q_B^b \times \{\beta_B\} \times Q_B^r)$ is the set of transitions.

The set of ready (resp. busy) states Q_B^r (resp. Q_B^b) is the set of states s.t. the component is ready (resp. not ready) to perform an action. Component B (i) has actions in set Act_B , which are possibly shared with some of the other components, (ii) has an internal action β_B s.t. $\beta_B \notin Act_B$ which models internal computations of component B , and (iii) alternates moving from a ready state to a busy state and from a busy state to a ready state. Note that busy states permit the modelling of distributed (decentralized) execution of components: after an interaction, components stay in busy states until the internal computation related to the interaction terminates (after which they get ready for the next interaction and so on). The state of components is only modified by their internal actions; other actions are dedicated to synchronisation.

We assume that each component $B_i \in \mathbf{B}$ is defined by the LTS $(Q_{B_i}, Act_{B_i} \cup \{\beta_{B_i}\}, \rightarrow_{B_i})$ where Q_{B_i} has a partition $\{Q_{B_i}^r, Q_{B_i}^b\}$ of ready and busy states.

Definition 2 (Behavior of a scheduler). *The behavior of a scheduler S is an LTS $(Q_S, Act_S, \rightarrow_S)$ s.t.:*

- Q_S is the set of states,
- $Act_S = Act_S^\gamma \cup Act_S^\beta$ is the set of actions, where $Act_S^\gamma = \{a \in Int \mid \text{mng}(a) = S\}$ and $Act_S^\beta = \{\beta_i \mid B_i \in \text{scp}(S)\}$,
- $\rightarrow_S \subseteq Q_S \times Act_S \times Q_S$ is the set of transitions.

$Act_S^\gamma \subseteq Int$ is the set of interactions managed by S , and Act_S^β is the set of internal actions of the components involved in an action managed by S .

In the following, we assume that each scheduler $S_j \in \mathbf{S}$ is defined by the LTS $(Q_{S_j}, Act_{S_j}, \rightarrow_{S_j})$ where $Act_{S_j} = Act_{S_j}^\gamma \cup Act_{S_j}^\beta$; as per Definition 2. The coordination of interactions of the system i.e., the interactions in Int , is distributed among schedulers. Actions of schedulers consist of interactions of the system. Since one scheduler is associated with each interaction, schedulers manage disjoint sets of interactions (i.e., $\forall S_i, S_j \in \mathbf{S}. S_i \neq S_j \implies Act_{S_i}^\gamma \cap Act_{S_j}^\gamma = \emptyset$). Intuitively, when a scheduler executes an interaction, it triggers the execution of the associated actions on the involved components. Moreover, when a component executes an internal action, it triggers the execution of the corresponding action on the associated schedulers and also sends the updated state of the component to the associated schedulers, that is, the component sends a message including its current state to the schedulers. Note, by construction, schedulers are always ready to receive such a state update.

Remark 1. Since components send their updated states to the associated schedulers, the current state of a scheduler contains the last state of each component in its scope.

Definition 3 (Shared component). $\mathbf{B}_s = \{B \in \mathbf{B} \mid |\{S \in \mathbf{S} \mid B \in \text{scp}(S)\}| \geq 2\}$.

A shared component is in the scope of more than one scheduler. Thus, the execution of its actions are managed by more than one scheduler. The global execution of the system can be described as the parallel execution of interactions managed by the schedulers.

Definition 4 (Global behavior). *The system behavior is the LTS $(Q, GAct, \rightarrow)$ where:*

- $Q \subseteq \bigotimes_{i=1}^{|\mathbf{B}|} Q_i \times \bigotimes_{j=1}^{|\mathbf{S}|} Q_{S_j}$ is the set of states consisting of the states of schedulers and components,
- $GAct \subseteq 2^{Int} \cup \bigcup_{i=1}^{|\mathbf{B}|} \{\beta_i\} \setminus \{\emptyset\}$ is the set of possible global actions of the system consisting of either several interactions and/or several internal actions (several interactions can be executed concurrently by the system),
- $\rightarrow \subseteq Q \times GAct \times Q$ is the transition relation defined as the smallest set abiding by the following rule. A transition is a move from state $(q_1, \dots, q_{|\mathbf{B}|}, q_{s_1}, \dots, q_{s_{|\mathbf{S}|}})$ to state $(q'_1, \dots, q'_{|\mathbf{B}|}, q'_{s_1}, \dots, q'_{s_{|\mathbf{S}|}})$ on global actions in set $\alpha \cup \beta$, where $\alpha \subseteq Int$ and $\beta \subseteq \bigcup_{i=1}^{|\mathbf{B}|} \{\beta_i\}$, noted $(q_1, \dots, q_{|\mathbf{B}|}, q_{s_1}, \dots, q_{s_{|\mathbf{S}|}}) \xrightarrow{\alpha \cup \beta} (q'_1, \dots, q'_{|\mathbf{B}|}, q'_{s_1}, \dots, q'_{s_{|\mathbf{S}|}})$, whenever the following conditions hold:
 - $C_1: \forall i \in [1..|\mathbf{B}|]. |(\alpha \cap Act_i) \cup (\{\beta_i\} \cap \beta)| \leq 1$,
 - $C_2: \forall a \in \alpha. (\exists S_j \in \mathbf{S}. \text{mng}(a) = S_j)$

$$\implies \left(q_{s_j} \xrightarrow{a}_{S_j} q'_{s_j} \wedge \forall B_i \in \text{inv}(a). q_i \xrightarrow{a \cap Act_i}_{B_i} q'_i \right),$$
 - $C_3: \forall \beta_i \in \beta. q_i \xrightarrow{\beta_i}_{B_i} q'_i \wedge \forall S_j \in \mathbf{S}. B_i \in \text{scp}(S_j). q_{s_j} \xrightarrow{\beta_i}_{S_j} q'_{s_j}$,
 - $C_4: \forall B_i \in \mathbf{B} \setminus \text{inv}(\alpha \cup \beta). q_i = q'_i$,
 - $C_5: \forall S_j \in \mathbf{S} \setminus \text{mng}(\alpha). q_{s_j} = q'_{s_j}$.

where functions *inv* and *mng* are extended to sets of interactions and internal actions.

The system components execute according to the schedulers decisions.

- C_1 states that a component performs at most one execution step at a time. Executed global actions $(\alpha \cup \beta)$ contains at most one interaction involving each component.
- *Condition C_2* states that whenever an interaction a managed by scheduler S_j is executed, a is enabled in S_j and the corresponding action (in $a \cap Act_i$) is enabled in each component involved in this interaction.
- *Condition C_3* states that internal actions are executed whenever they are enabled in the corresponding components. Schedulers are aware of internal actions of components in their scope. This results in transferring the updated state to the schedulers.
- *Conditions C_4 and C_5* state that the components and the schedulers not involved in an interaction remain in the same state.

Remark 2. The operational description of a CBS is usually more detailed. The execution of conflicting interactions in schedulers needs first to be authorized by a conflict-resolution module which guarantees that two conflicting interactions

are not executed at the same time. Moreover, schedulers follow the (possible) priority rules among the interactions, i.e., in the case of two or more enabled interactions (interactions, which are ready to be executed by schedulers), those with higher priority are allowed to be executed. Since we only deal with execution traces, we assume that these are correct w.r.t. the conflicts and priorities. Therefore, defining the other modules is out of our scope. Moreover, schedulers could interact together as part of some coordination protocol, but our model does not account for it.

Definition 5 (Monitoring hypothesis). *The behavior of the CBS under scrutiny can be modeled as an LTS as per Definition 4.*

3.2 Traces

Running the system produces a trace. Intuitively, a trace is the sequence of traversed states of the system, from some initial state and following the transition relation of the LTS of the system. For the sake of simplicity and for our monitoring purposes, the states of schedulers are irrelevant in the trace and thus we restrict the system states to states of the components.

We consider a CBS consisting of a set \mathbf{B} of components (as per Definition 1) and a set \mathbf{S} of schedulers (as per Definition 2) with the global behavior as per Definition 4.

Definition 6 (Trace). *A trace is a sequence $(q_1^0, \dots, q_{|\mathbf{B}|}^0) \cdot (\alpha^0 \cup \beta^0) \cdot (q_1^1, \dots, q_{|\mathbf{B}|}^1) \cdots (q_1^k, \dots, q_{|\mathbf{B}|}^k) \cdots$, s.t. $q_1^0, \dots, q_{|\mathbf{B}|}^0$ are the initial states of components $B_1, \dots, B_{|\mathbf{B}|}$ and $\forall i \in [0..k-1]. (q_1^i, \dots, q_{|\mathbf{B}|}^i) \xrightarrow{\alpha^i \cup \beta^i} (q_1^{i+1}, \dots, q_{|\mathbf{B}|}^{i+1})$, where \rightarrow is the transition relation of the global system and scheduler states are discarded.*

Since a trace t has partial states where at least one component is busy with its internal computation, t is referred to as a *partial trace*. Although the partial trace of the system exists, it is not observable because it would require a perfect observer having simultaneous access to the states of the components. Introducing such an observer in the system would require all components to synchronize, and would defeat the purpose of building a distributed system. Instead, we shall instrument the system to observe the sequence of states through schedulers.

In the sequel, we consider a partial trace $t = (q_1^0, \dots, q_{|\mathbf{B}|}^0) \cdot (\alpha^0 \cup \beta^0) \cdot (q_1^1, \dots, q_{|\mathbf{B}|}^1) \cdots$, as per Definition 6. Each scheduler $S_j \in \mathbf{S}$, observes a local partial trace $s_j(t)$ which consists in the sequence of state-evolutions of the components it manages.

Definition 7 (Observable local partial-trace). *The local partial-trace $s_j(t)$ observed by scheduler S_j is defined on the partial trace t as follows:*

$$\begin{aligned}
& - s_j \left(\left(q_1^0, \dots, q_{|\mathbf{B}|}^0 \right) \right) = \left(q_1^0, \dots, q_{|\mathbf{B}|}^0 \right), \text{ and} \\
& - s_j (t \cdot (\alpha \cup \beta) \cdot q) = \begin{cases} t & \text{if } S_j \notin \text{mng}(\alpha) \wedge (\text{inv}(\beta) \cap \text{scp}(S_j) = \emptyset) \\ t \cdot \gamma \cdot q' & \text{otherwise} \end{cases}
\end{aligned}$$

where

- $q = (q_1, \dots, q_{|\mathbf{B}|})$,
- $\gamma = (\alpha \cap \{a \in \text{Int} \mid \text{mng}(a) = S_j\}) \cup (\beta \cap \{\beta_i \mid B_i \in \text{scp}(S_j)\})$
- $q' = (q'_1, \dots, q'_{|\mathbf{B}|})$ with $q'_i = \begin{cases} \text{last}(s_j(t))[i] & \text{if } B_i \in \overline{\text{inv}(\gamma)} \cap \text{scp}(S_j), \\ q_i & \text{if } B_i \in \text{inv}(\gamma) \cap \text{scp}(S_j), \\ ? & \text{otherwise } (B_i \notin \text{scp}(S_j)). \end{cases}$

We assume that the initial system state is observable by all schedulers. An interaction $a \in \text{Int}$ is observable by scheduler S_j if S_j manages the interaction (i.e., $S_j \in \text{mng}(a)$). Moreover, an internal action β_i , $i \in [1..|\mathbf{B}|]$, is observable by scheduler S_j if B_i is in the scope of S_j . The state observed after an observable interaction or internal action consists of the states of components in the scope of S_j , i.e., a state $(q_1, \dots, q_{|\mathbf{B}|})$ where q_i is the new state of component B_i if $B_i \in \text{scp}(S_j)$ and ? otherwise.

4 Efficient Construction of the Computation Lattice

We define how a global observer constructs on-the-fly a computation lattice representing the possible global traces compatible with the local partial-traces observable by schedulers. Since schedulers do not interact directly, the execution of an interaction by one scheduler seems to be concurrent with the execution of all interactions by other schedulers. Nevertheless, if scheduler S_j manages interaction a and scheduler S_k manages interaction b s.t. a shared component $B_i \in \mathbf{B}_s$ is involved in a and b , i.e., $B_i \in \text{inv}(a) \cap \text{inv}(b)$, the execution of interactions a and b are causally related. In other words, there exists only one possible ordering of a and b and they could not have been executed concurrently. Ignoring the actual ordering of a and b would result in retrieving inconsistent global states (i.e., states that do not belong to the system). To find out the actual ordering and obtain the local partial-traces, one needs instrumenting the system by adding controllers to the schedulers and to the shared components. Each time a scheduler executes an interaction, the involved components are notified by the scheduler to execute their corresponding actions. Moreover, the controller of the scheduler updates its local clock and notifies the controller of the shared components involved in the interaction by sending its vector clock. Whenever a shared component executes its internal action β , schedulers with the shared component in their scope are notified by receiving the updated state. Moreover, the vector clock stored in the controller of the shared component is sent to the controller of the associated schedulers. Consequently, schedulers with a shared component in their scope exchange their vector clocks through the shared component. Such an instrumentation is described in [29] but omitted for space reasons.

Intuitively, for scheduler S_j , the execution of an interaction (labeled by a vector clock), or notification by the internal action of a component which the

execution of its latest action has been managed by scheduler S_j , is defined as an *event* of scheduler S_j . For a partial trace t , the sequence of events of scheduler S_j is denoted by $\text{event}(s_j(t))$.

4.1 Computation Lattice

The computation lattice is represented implicitly using vector clocks. The construction mainly performs the two following operations: (i) *creations of new nodes* and (ii) *updates* of existing nodes in the lattice. The observer receives two sorts of events: events related to the execution of an interaction in Int , referred to as *action events*, and events related to internal actions referred to as *update events*. (Recall that internal actions carry the state of the component that has performed the action – the state is transmitted to the observer by the controller that is notified of this action. See Sect. 3). Hence, the set of action events is defined as $E_a = \text{Int} \times \text{VC}$ with VC the set of vector clocks, and the set of update events is defined as $E_\beta = \bigcup_{i \in [1, |\mathbf{B}|]} (\{\beta_i\} \times Q_i)$. Action events lead to the creation of new nodes in the direction of the scheduler emitting the event while update events complete the information in the nodes of the lattice related to the state of the component related to the event. The set of all events is $E = E_\beta \cup E_a$. Since the received events are not totally ordered (because of communication delay), we construct the computation lattice based on the vector clocks attached to the received events. Note, we assume that the events received from a scheduler are totally ordered.

We first adapt the notion of computation lattice to CBSs.

Definition 8 (Computation lattice). *A computation lattice \mathcal{L} is a tuple $(N, \text{Int}, \succ\!\!\!\rightarrow)$, where:*

- $N \subseteq Q^l \times \text{VC}$ is the set of nodes, with VC the set of vector clocks and $Q^l = \bigotimes_{i=1}^{|\mathbf{B}|} \left(Q_i^r \cup \left\{ \perp_i^j \mid S_j \in \mathbf{S} \wedge B_i \in \text{scp}(S_j) \right\} \right)$,
- Int is the set of multi-party interactions as defined in Sect. 3.1,
- $\succ\!\!\!\rightarrow = \{(\eta, a, \eta') \in N \times \text{Int} \times N \mid a \in \text{Int} \wedge \eta \mapsto \eta' \wedge \eta.\text{state} \xrightarrow{a} \eta'.\text{state}\}$,

where $\succ\!\!\!\rightarrow$ is the extended happened-before relation, which is labeled by the set of multi-party interactions and $\eta.\text{state}$ refers to the state of node η .

Intuitively, a computation lattice consists of a set of partially connected nodes, where each node is a pair, consisting of a system state and a vector clock. A system state consists of the states of all components. The state of a component is either a ready state or a busy state (as per Definition 1). We represent a busy state of component B_i , by \perp_i^j which shows that component B_i is busy to finish its latest action which has been managed by scheduler S_j . A computation lattice \mathcal{L} initially consists of an initial node $\text{init}_{\mathcal{L}} = (\text{init}, (0, \dots, 0))$, where init is the initial state of the system and $(0, \dots, 0)$ is a vector clock where all the clocks associated with the schedulers are zero. The set of nodes of \mathcal{L} is denoted by $\mathcal{L}.\text{nodes}$, and for a node $\eta = (q, \text{vc}) \in \mathcal{L}.\text{nodes}$, $\eta.\text{state}$ denotes q and $\eta.\text{clock}$ denotes vc . If (i) the event of node η happened before the events of node η' ,

that is $\eta'.clock > \eta.clock$ and $\eta \rightsquigarrow \eta'$, and (ii) the states of η and η' follow the global behavior of the system (Definition 4) in the sense that the execution of an interaction $a \in Int$ from the state of η brings the system to the state of η' , that is $\eta.state \xrightarrow{a} \eta'.state$, then in the computation lattice it is denoted by $\eta \rightsquigarrow^a \eta'$ or by $\eta \rightsquigarrow \eta'$ when clear from context.

Two nodes η and η' of the computation lattice \mathcal{L} are said to be concurrent if neither $\eta.clock > \eta'.clock$ nor $\eta'.clock > \eta.clock$. For two concurrent nodes η and η' if there exists a node η'' s.t. $\eta'' \rightsquigarrow \eta$ and $\eta'' \rightsquigarrow \eta'$, then node η'' is said to be the *meet* of η and η' denoted by $meet(\eta, \eta', \mathcal{L}) = \eta''$.

4.2 Intermediate Operations

We consider a computation lattice \mathcal{L} (Definition 8). A received event either modifies \mathcal{L} or is kept for later in a queue. Action events extend \mathcal{L} using operator *extend* (Definition 9), and update events update the existing nodes of \mathcal{L} by adding the missing state information into them using operator *update* (Definition 12). By extending the lattice with new nodes, one needs to further complete the lattice by computing the joins of created nodes (Definition 11) with existing ones to complete the set of possible global states and traces.

Extension of the Lattice. We define a function to extend a node of the lattice with an action event which takes as input a node and an action event and outputs a new node.

Definition 9 (Node extension). *Given a node $\eta = (q, vc) \in Q^l \times VC$ and an action event $e = (a, vc') \in E_a$, function *extend* : $(Q^l \times VC) \times E_a \rightarrow Q^l \times VC$ is defined as follows: $extend(\eta, e) =$*

$$\begin{cases} (q', vc') & \text{if } \exists j \in [1..|\mathbf{S}|] . (vc'[j] = vc[j] + 1 \wedge \\ & \forall j' \in [1..|\mathbf{S}|] \setminus \{j\} . vc'[j'] = vc[j']) \\ \text{undefined} & \text{otherwise ;} \end{cases}$$

with $\forall i \in [1..|\mathbf{B}|] . q'[i] = \begin{cases} \perp_i^k & \text{if } B_i \in \text{inv}(a), \text{ where } k = \text{mng}(a).index, \\ q[i] & \text{otherwise.} \end{cases}$

Node η is said to be *extendable* by event e if $extend(\eta, e)$ is defined. Node $\eta = (q, vc)$ represents a global state of the system and extensibility of η by action event $e = (a, vc')$ means that from the global state q , scheduler $S_j = \text{mng}(a)$, could execute interaction a . State \perp_i^k indicates that component B_i is busy and being involved in a global action which has been executed (managed) by scheduler S_k for $k \in [1..|\mathbf{S}|]$.

We say that \mathcal{L} is extendable by action event e if there exists a node $\eta \in \mathcal{L}.nodes$ s.t. $extend(\eta, e)$ is defined.

Property 1. $\forall e \in E_a . |\{\eta \in \mathcal{L}.nodes \mid \exists \eta' \in Q^l \times VC . \eta' = extend(\eta, e)\}| \leq 1$.

Property 1 states that for any action event e , there exists at most one node in the lattice for which function *extend* is defined (meaning that \mathcal{L} can be extended by event e from that node). We define a relation between two vector clocks to

distinguish the concurrent execution of two interactions s.t. both could happen from a specific global system state.

Definition 10 (Relation $\mathcal{J}_{\mathcal{L}}$). $\mathcal{J}_{\mathcal{L}} = \{(vc, vc') \in VC \times VC \mid \exists! k \in [1..|\mathbf{S}|]. vc[k] = vc'[k] + 1 \wedge \exists! l \in [1..|\mathbf{S}|]. vc'[l] = vc[l] + 1 \wedge \forall j \in [1..|\mathbf{S}|] \setminus \{k, l\}. vc[j] = vc'[j]\}$.

For two vector clocks vc and vc' to be in $\mathcal{J}_{\mathcal{L}}$, they should agree on all but two clock values related to two schedulers of indexes k and l . On one index, the value of one vector clock is equal to the value of the other vector clock plus 1, and the converse on the other index. Intuitively, $(\eta.clock, \eta'.clock) \in \mathcal{J}_{\mathcal{L}}$ means that η and η' are associated with two concurrent events (caused by the execution of two interactions managed by different schedulers) that both could happen from a unique global system state, which is the meet of η and η' (see Property 2).

Property 2. $\forall \eta, \eta' \in \mathcal{L}.nodes. (\eta.clock, \eta'.clock) \in \mathcal{J}_{\mathcal{L}} \Rightarrow \text{meet}(\eta, \eta', \mathcal{L}) \in \mathcal{L}.nodes.$

The join of two nodes is defined as follows.

Definition 11 (Join node). For two nodes $\eta, \eta' \in \mathcal{L}.nodes$ s.t. $(\eta.clock, \eta'.clock) \in \mathcal{J}_{\mathcal{L}}$, the join of η and η' , denoted by $\text{join}(\eta, \eta', \mathcal{L}) = \eta''$, is the node defined as follows:

- $\forall i \in [1..|\mathbf{B}|]. \eta''.state[i] = \begin{cases} \eta.state[i] & \text{if } \eta.state[i] \neq \eta_m.state[i], \\ \eta'.state[i] & \text{otherwise;} \end{cases}$
- $\eta''.clock = \max(\eta.clock, \eta'.clock);$ where $\eta_m = \text{meet}(\eta, \eta', \mathcal{L})$.

According to Property 2, for two nodes η and η' in relation $\mathcal{J}_{\mathcal{L}}$, their meet node exists in the lattice. The state of the join of η and η' is defined by comparing their states and the state of their meet. Since two nodes in relation $\mathcal{J}_{\mathcal{L}}$ are concurrent, the state of component B_i for $i \in [1..|\mathbf{B}|]$ in nodes η and η' is either equal to the state of component B_i in their meet, or only one of the nodes η and η' has a state different from their meet (components can not be both involved in two concurrent executions). The join node of two nodes η and η' takes into account the latest changes of the state of the nodes η and η' compared to their meet. Note that $\text{join}(\eta, \eta', \mathcal{L}) = \text{join}(\eta', \eta, \mathcal{L})$, because join is defined for nodes whose clocks are in relation $\mathcal{J}_{\mathcal{L}}$.

Update of the Lattice. We define a function to update a node of the lattice which takes as input a node and an update event and outputs the updated version of the input node.

Definition 12 (Node update). Given a node $\eta = ((q_1, \dots, q_{|\mathbf{B}|}), vc)$ and an update event $e = (\beta_i, q'_i) \in E_{\beta}$ with $i \in [1..|\mathbf{B}|]$, which is sent by scheduler S_k with $k \in [1..|\mathbf{S}|]$, function $\text{update} : (Q^l \times VC) \times E_{\beta} \rightarrow Q^l \times VC$ is defined as follows:

$$\text{update}(\eta, e) = ((q_1, \dots, q_{i-1}, q''_i, q_{i+1}, \dots, q_{|\mathbf{B}|}), vc), \text{ with } q''_i = \begin{cases} q'_i & \text{if } q_i = \perp_i^k, \\ q_i & \text{otherwise.} \end{cases}$$

An update event (β_i, q'_i) contains an updated state of some component B_i . By updating a node η in the lattice with an update event, which is sent from scheduler S_k , we update the partial state associated to η by adding the state information of that component, if the state of component B_i associated to node η is \perp_i^k . Intuitively it means that a busy state resulting of the execution of an action managed by scheduler S_k can only be replaced by a ready state sent by S_k . Updating node η does not modify vc .

Buffering Events. The reception of an action or update event might not always lead to extending or updating the current computation lattice. Due to communication delay, an event which has happened before another event might be received later by the observer. It is necessary for the construction of the lattice to use events in a specific order. Events not in the desired order must be kept in a waiting queue to be used later. For example, such a situation occurs when receiving action event e s.t. function `extend` is not defined over e and none of the existing nodes of the lattice. Event e must be kept in the queue until obtaining another configuration of the lattice in which function `extend` is defined. Moreover, an update event e' referring to an internal action of component B_i is kept in the queue if there exists an action event e'' in the queue s.t. component B_i is involved in e'' , because we can not update the nodes of the lattice with an update event associated to an execution, which is not yet taken into account in the lattice.

Definition 13 (Queue κ). *A queue of events is a finite sequence of events in E . Moreover, for a non-empty queue $\kappa = e_1 \cdot e_2 \cdots e_r$, $\text{remove}(\kappa, e) = \kappa(1 \cdots z - 1) \cdot \kappa(z + 1 \cdots r)$ with $e = e_z \in \{e_1, e_2, \dots, e_r\}$. Moreover, events in the queue are picked up in the same order as they have been stored in the queue (FIFO queue).*

4.3 Algorithms for Constructing the Computation Lattice

We define an algorithm based on the above definitions to construct the computation lattice based on the received events. The algorithm consists of a main procedure (see Algorithm 1) and several sub-procedures. The algorithm defines and uses a lattice (Definition 8, global variable \mathcal{L}) and a queue (Definition 13, global variable κ).

For an action event $e \in E_a$ with $e = (a, vc)$, $e.action$ denotes interaction a and $e.clock$ denotes vector clock vc . For an update event $e \in E_\beta$ with $e = (\beta_i, q_i)$, $e.index$ denotes index i .

After the reception of each event e from a controller of a scheduler, $\text{MAKE}(e) = \text{MAKE}(e, \text{false})$ is called. In the sequel, we describe each procedure.

Algorithm 1. MAKE

Global variables: \mathcal{L} initialized to $init_{\mathcal{L}}$,
1: κ initialized to ϵ ,
2: V initialized to $(0, \dots, 0)$.
3: **procedure** MAKE(e , *from-queue*)
4: **if** $e \in E_a$ **then**
5: ACTIONEVENT(e , *from-queue*)
6: **else if** $e \in E_b$ **then**
7: UPDATEEVENT(e , *from-queue*)
8: **end if**
9: **end procedure**

cedure MAKE uses two sub-procedures, ACTIONEVENT and UPDATEEVENT. MAKE updates the global variables.

ActionEvent (Algorithm 2). Procedure ACTIONEVENT takes as input an action event e and a boolean parameter. Procedure ACTIONEVENT modifies global variables \mathcal{L} and κ . ACTIONEVENT has a local boolean variable named *lattice-extend*, which is true when an input action event could extend the lattice (i.e., the current computation lattice is extendable by the input action event) and false otherwise. By iterating over the existing nodes, ACTIONEVENT checks if there exists a node η in $\mathcal{L}.nodes$ s.t. function *extend* is defined over event e and node η (Definition 9). If such a node η is found, ACTIONEVENT creates the new node $extend(\eta, e)$, adds it to the set of the nodes of the lattice, invokes procedure MODIFYQUEUE, and stops iteration. Otherwise, ACTIONEVENT invokes procedure MODIFYQUEUE and terminates. In the case of extending the lattice by a new node, it is necessary to create the (possible) join nodes. To this end, in Line 15 procedure JOINS is called to evaluate the current lattice and create the join nodes. For optimization purposes, REMOVEEXTRANODES is then called to eliminate unnecessary nodes that represent past system states. After making the join nodes and (possibly) reducing the size, if the input action event is not picked from the queue, ACTIONEVENT invokes procedure CHECKQUEUE in Line 18, otherwise it terminates.

Algorithm 2. ACTIONEVENT

1: **procedure** ACTIONEVENT(e , *from-queue*)
2: *lattice-extend* \leftarrow **false**
3: **for all** $\eta \in \mathcal{L}.nodes$ **do**
4: **if** $\exists \eta' \in Q^t \times VC. \eta' = extend(\eta, e)$ **then**
5: $\mathcal{L}.nodes \leftarrow \mathcal{L}.nodes \cup \{\eta'\}$
6: MODIFYQUEUE(e , *from-queue*, **true**)
7: *lattice-extend* \leftarrow **true**
8: **break**
9: **end if**
10: **end for**
11: **if** \neg *lattice-extend* **then**
12: MODIFYQUEUE(e , *from-queue*, **false**)
13: **return**
14: **end if**
15: JOINS(\cdot)
16: REMOVEEXTRANODES()
17: **if** \neg *from-queue* **then**
18: CHECKQUEUE(\cdot)
19: **end if**
20: **end procedure**

Make (Algorithm 1). Procedure MAKE takes two parameters as input: an event e and a boolean variable *from-queue*. Parameters e and *from-queue* vary based on the type of event e . Boolean variable *from-queue* is true when the input event e is picked up from the queue and false otherwise (i.e., event e is received from a controller of a scheduler). Pro-

UpdateEvent (Algorithm 3). Recall that an update event e contains the state update of some component B_i with $i \in [1, n]$ ($e.index = i$). UPDATEEVENT takes as input an update event e and a boolean value associated with parameter *from-queue*. UPDATEEVENT modifies global variables \mathcal{L} and κ . First, UPDATEEVENT checks the events in the queue. If there exists an action event e' in the queue s.t. component B_i is involved in $e'.action$, UPDATEEVENT adds update event e to the queue using MODIFYQUEUE and terminates. Indeed, one can not update the

nodes of the lattice with an update event associated to an execution, which is not yet taken into account in the lattice. If no action event in the queue concerns component B_i , UPDATEEVENT updates all the nodes of the lattice (Lines 8–10) according to Definition 12. Finally, the input update event is removed from the queue if it is picked from the queue, using MODIFYQUEUE.

ModifyQueue takes as input an event e and boolean variables *from-queue* and *event-is-used*. Procedure MODIFYQUEUE adds (resp. removes) event e to (resp. from) queue κ . If event e is picked up from the queue (i.e., *from-queue* = **true**) and e is used in the algorithm to extend or update the lattice (i.e., *event-is-used* = **true**), event e is removed from the queue. Moreover, if event e is not picked up from the queue and it is not used in the algorithm, event e is stored in the queue.

Joins extends \mathcal{L} in such a way that all the possible joins have been created. First, procedure JCOMPUTE is invoked to compute relation $\mathcal{J}_{\mathcal{L}}$ (Definition 10) among the existing nodes of the lattice and then creates the join nodes and adds them to the set of the nodes. Then, after the creation of the join of two nodes η and η' , $(\eta.clock, \eta'.clock)$ is removed from $\mathcal{J}_{\mathcal{L}}$. It is necessary to compute $\mathcal{J}_{\mathcal{L}}$ again after the creation of joins, because new nodes can be in $\mathcal{J}_{\mathcal{L}}$. This process terminates when $\mathcal{J}_{\mathcal{L}}$ is empty.

Jcompute computes relation $\mathcal{J}_{\mathcal{L}}$ by pairwise iteration over all the nodes of the lattice and checks if the vector clocks of any two nodes satisfy the conditions in Definition 10. The pair of vector clocks satisfying the above conditions are added to $\mathcal{J}_{\mathcal{L}}$.

Algorithm 3. UPDATEEVENT

```

1: procedure UPDATEEVENT( $e, from\_queue$ )
2:   for all  $e' \in \kappa$  do
3:     if  $e' \in E_a \wedge e.index \in inv(e'.action)$  then
4:       MODIFYQUEUE( $e, from\_queue, false$ )
5:       return
6:     end if
7:   end for
8:   for all  $\eta \in \mathcal{L}.nodes$  do
9:      $\eta \leftarrow update(\eta, e)$ 
10:  end for
11:  MODIFYQUEUE( $e, from\_queue, true$ )
12: end procedure

```

CheckQueue recalls the events stored in the queue $e \in \kappa$ and then executes MAKE($e, true$), to check whether the conditions for taking them into account to update the lattice hold. CHECKQUEUE checks the events in the queue until none of the events in the queue can be used either to extend or to update the lattice. To this end, before checking queue κ , a copy of queue κ is stored in κ' , and after iter-

ating all the events in queue κ , the algorithm checks the equality of current queue and the copy of the queue before checking. If the current queue κ and copied queue κ' have the same events, it means that none of the events in queue κ has been used (thus removed), therefore the algorithm stops checking the queue again by breaking the loop. Note, when the algorithm is iterating over the events in the queue, i.e., when the value of variable *from-queue* is true, it is not necessary to iterate over the queue again (Algorithm 2, Line 17).

RemoveExtraNodes removes the extra nodes of the lattice. Since our online algorithm is used for runtime monitoring purposes, each node n represents the evaluation of system execution up to node n . Hence, the nodes which reflect the

state of the system in the past are not valuable for the runtime monitor. For this, after extending the lattice by an action event, procedure REMOVEEXTRANODES is called to eliminate some (possibly existing) nodes of the lattice. A node in the lattice can be removed if the lattice no longer can be extended from that node. Having two nodes of the lattice η and η' s.t. every clock in the vector clock of η' is strictly greater than the respective clock of η , one can remove node η . This is due to the fact that the algorithm never receives an action event which could have extended the lattice from η where the lattice has already took into account the occurrence of an event which has greater clock stamps than $\eta.clock$.

5 Properties of the Constructed Lattice

We give the properties of the lattice constructed in the previous section.

5.1 Insensitivity to Communication Delay

Algorithm MAKE can be defined over a sequence of events received by the observer $\zeta = e_1 \cdot e_2 \cdot e_3 \cdots e_z \in E^*$ by applying it sequentially from e_1 to e_z with the initial lattice $init_{\mathcal{L}}$ and an empty queue.

Proposition 1 (Insensitivity to the reception order). $\forall \zeta, \zeta' \in E^*, \forall S_j \in \mathbf{S} \cdot \zeta \downarrow_{S_j} = \zeta' \downarrow_{S_j} \implies \text{MAKE}(\zeta) = \text{MAKE}(\zeta')$, where $\zeta \downarrow_{S_j}$ is the projection of ζ on scheduler S_j which results the sequence of events generated by S_j .

Proposition 1 states that different ordering of the events does not affect the output result of Algorithm MAKE. Note, Proposition 1 assumes that all events in ζ and ζ' can be distinguished. For a sequence of events $\zeta \in E^*$, $\text{MAKE}(\zeta).lattice$ denotes the constructed computation lattice \mathcal{L} by algorithm MAKE.

5.2 Correctness of Lattice Construction

Computation lattice \mathcal{L} has a *frontier* node, which is the node with the greatest vector clock. A path of the constructed computation lattice \mathcal{L} is a sequence of causally related nodes of the lattice, starting from the initial node and ending up in the frontier node.

Definition 14 (Set of the paths of a lattice). *The set of the paths of a constructed computation lattice \mathcal{L} is $\Pi(\mathcal{L}) = \left\{ \eta_0 \cdot \alpha_1 \cdot \eta_1 \cdot \alpha_2 \cdot \eta_2 \cdots \alpha_z \cdot \eta_z \mid \eta_0 = init_{\mathcal{L}} \wedge \forall r \in [1 \dots z] \cdot \left(\eta_{r-1} \xrightarrow{\alpha_r} \eta_r \vee (\exists N \subseteq \mathcal{L}.nodes \cdot \eta_{r-1} = \text{meet}(N, \mathcal{L}) \wedge \eta_r = \text{join}(N, \mathcal{L}) \wedge \forall \eta \in N \cdot \eta_{r-1} \xrightarrow{a_\eta} \eta \wedge \alpha_r = \bigcup_{\eta \in N} a_\eta \right) \right\}$, where the notions of meet and join are naturally extended to a set of nodes.*

A path is a sequence of nodes s.t. for each pair of adjacent nodes either (i) the prior node and the next node are related according to $\xrightarrow{\alpha}$ or (ii) the prior and the next node are the meet and the join of a set of existing nodes respectively.

A path from a meet node to the associated join node represents an execution of a set of concurrent interactions.

At runtime, the execution of such a system produces a partial trace $t = q^0 \cdot (\alpha^1 \cup \beta^1) \cdot q^1 \cdot (\alpha^2 \cup \beta^2) \cdot \dots \cdot (\alpha^k \cup \beta^k) \cdot q^k$ which consists of partial states and global actions (Definition 6). Due to the occurrence of concurrent interactions and internal actions, each partial trace can be represented as a set of compatible and possible partial traces.

Definition 15 (Compatible partial-traces of a partial trace). *The set of all compatible partial-traces of partial trace t is $\mathcal{P}(t) = \{t' \in Q \cdot (GAct \cdot Q)^* \mid \forall j \in [1 \dots |\mathbf{S}|], t' \downarrow_{S_j} = t \downarrow_{S_j} = s_j(t)\}$.*

Trace t' is compatible with trace t if the projection of both t and t' on scheduler S_j , for $j \in [1 \dots |\mathbf{S}|]$, results the local trace of scheduler S_j . In a partial trace, for each global action which consists of several concurrent interactions and internal actions of different schedulers, one can define different ordering of those concurrent interactions, each of which represents a possible execution of that global action. Consequently, several compatible partial-traces can be encoded from a partial trace.

Note that two compatible traces with only difference in the ordering of their internal actions are considered as a unique compatible trace. Two compatible traces of a partial trace differ if they have different ordering of interactions.

For monitoring purposes we need to represent the run of the system by a sequence of global states (recall that we consider properties over global states). For this, we extend the technique in [28], to define a function which takes as input a partial trace of the distributed system (i.e., a sequence of partial states) and outputs an equivalent global trace in which all the internal actions (β) are removed from the trace and instead the updated state after each internal action is used to complete the states of the partial trace.

Definition 16 (Function refine \mathcal{R}). *Function $\mathcal{R} : Q \cdot (GAct \cdot Q)^* \rightarrow Q \cdot (Int \cdot Q)^*$ is defined as $\mathcal{R}(init) = init$ and:*

$$\mathcal{R}(\sigma \cdot (\alpha \cup \beta) \cdot q) = \begin{cases} \mathcal{R}(\sigma) \cdot \alpha \cdot q & \text{if } \beta = \emptyset, \\ \text{map } [x \mapsto \text{upd}(q, x)] (\mathcal{R}(\sigma)) & \text{if } \alpha = \emptyset, \\ \text{map } [x \mapsto \text{upd}(q, x)] (\mathcal{R}(\sigma) \cdot \alpha \cdot q) & \text{otherwise;} \end{cases}$$

with $\text{upd} : Q \times (Q \cup 2^{Int}) \rightarrow Q \cup 2^{Int}$ defined as: $\text{upd}((q_1, \dots, q_{|\mathbf{B}|}), \alpha) = \alpha$, and $\text{upd}((q_1, \dots, q_{|\mathbf{B}|}), (q'_1, \dots, q'_{|\mathbf{B}|})) = (q''_1, \dots, q''_{|\mathbf{B}|})$, where $\forall k \in [1 \dots |\mathbf{B}|], q''_k = q_k$ if $(q_k \notin Q_k^b) \wedge (q'_k \in Q_k^b)$ and q'_k otherwise.

Function \mathcal{R} uses the state after internal actions in order to update the partial states using function upd .

By applying function \mathcal{R} to the set of compatible partial-traces $\mathcal{P}(t)$, we obtain a new set of global traces, which is (i) equivalent to $\mathcal{P}(t)$ (according to [28], Def. 7), (ii) internal actions are discarded in the presentation of each global trace and (iii) contains maximal global states that can be built with the information contained in the partial states observed so far. In Sect. 3.2 (Definition 7)

we define $\{s_1(t), \dots, s_{|\mathbf{S}|}(t)\}$, the set of observable local partial-traces of the schedulers obtained from partial trace t . From each local partial-trace we can obtain the sequences of events generated by the controller of each scheduler, s.t. the set of all the sequences of the events is $\{\text{event}(s_1(t)), \dots, \text{event}(s_{|\mathbf{S}|}(t))\}$ with $\text{event}(s_j(t)) \in E^*$ for $j \in [1..|\mathbf{S}|]$.

In the following, we define the set of all possible sequences of events that could be received by the observer.

Definition 17 (Events ordering). *Considering partial trace t , the set of all possible sequences of events that could be received by the observer is $\Theta(t) = \{\zeta \in E^* \mid \forall j \in [1..|\mathbf{S}|]. \zeta \downarrow_{S_j} = \text{event}(s_j(t))\}$.*

Events are received by the observer in any order compatible with the local events of schedulers.

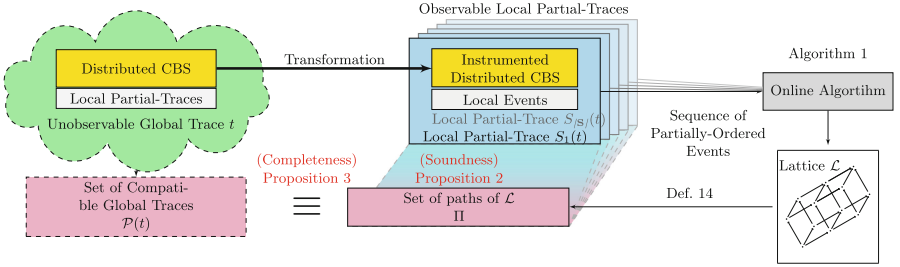


Fig. 1. Overview of the construction of the computation lattice.

Proposition 2 (Soundness). *Given a partial trace t as per Definition 6, we have:*

$$\forall \zeta \in \Theta(t), \forall \pi \in \Pi(\text{MAKE}(\zeta).lattice), \forall j \in [1..|\mathbf{S}|]. \pi \downarrow_{S_j} = \mathcal{R}(s_j(t)).$$

Proposition 2 states that the projection of all paths in the lattice on a scheduler S_j for $j \in [1..|\mathbf{S}|]$ results in the refined local partial-trace of scheduler S_j . The following proposition states the correctness of the construction in the sense that applying Algorithm MAKE to a sequence of observed events (i.e., $\zeta \in \Theta$) at runtime, results a computation lattice which encodes a set of the sequences of global states, s.t. each sequence represents a global trace of the system.

Proposition 3 (Completeness). *Given a partial trace t as per Definition 6, we have:*

$$\forall \zeta \in \Theta(t), \forall t' \in \mathcal{P}(t), \exists! \pi \in \Pi(\text{MAKE}(\zeta).lattice). \pi = \mathcal{R}(t').$$

π is said to be the associated path of the compatible partial-trace t' . Applying algorithm MAKE to any sequence of events constructs a computation lattice whose set of paths consists on all the compatible global traces.

Figure 1 depicts an overview of our approach.

6 Related Work

The problem of reconstructing a global behavior from local observations/behaviors has been investigated in several settings. In the setting of choreographies, the approach in [25] reconstructs global graphs from (local) communicating finite-state machines. More recently, in the setting of program replay, the approach in [24] introduces causal-consistent replay to record the execution of a concurrent program and reproduce a misbehavior as well as its causes inside a debugger.

In the following, we focus our comparison on the research efforts that contributed to the distribution of the monitoring process. The approaches in [4, 12, 14, 15] define algorithms for decentralized monitoring for distributed systems with a global clock. In comparison, we target asynchronous distributed CBSs with a partial-state semantics, where global states are not available at runtime. Hence, instead of having a global trace at runtime, we deal with a set of compatible partial traces which could have happened at runtime. The approach in [5] detects and analyzes synchronous distributed systems faults in a centralized manner using local LTL properties evaluated with local traces. In our setting, global properties can not be projected and checked on individual components nor on individual schedulers. Thus, local traces can not be directly used for verifying properties. In [31], the authors designed a method for monitoring safety properties in distributed systems relying on the existing communication among processes. Compared to [31], our algorithm is sound, in the sense that we reconstruct the behavior of the distributed system based on all possible partial-traces of the distributed system. In our work, each trace could have happened as the actual trace of the system, and could have generated the same events. The approach in [26] monitors LTL properties on finite executions of a distributed system. In comparison, our approach is tailored to and leverages the structure of CBSs; traces are defined over partial states and are obtained from a generic semantic model of CBSs.

There are several approaches dedicated to the monitoring of CBSs: [13, 21] for the correctness of reconfigurations of Fractal [11] components, [17] and [28] for the runtime verification of functional properties on respectively sequential and multi-threaded BIP [2] CBSs, [8] for the runtime checking of local behaviors specified as quantitative properties. However, these approaches do not handle fully concurrent components and they either assume a global clock or a shared memory. This is for instance the case with our previous work on monitoring multithreaded CBSs [28] where we assume a global clock shared by the threads used to execute the components.

7 Conclusions

Conclusions. We present a technique that enables the monitoring on distributed CBSs, where the interactions are partitioned among a set of distributed schedulers. Each scheduler is in charge of execution of a subset of interactions.

The execution of each interaction triggers the actions of the components involved in the interaction. Our technique consists in (i) transforming the system to generate events associated to partial trace local to each scheduler, (ii) synthesizing a centralized observer which collects the local events of all schedulers (iii) reconstructing on-the-fly the possible orderings of the received events which forms a computation lattice. Our technique leverages the nature of distributed CBSs in that it uses components shared by several interactions to infer causality relations between events. The constructed lattice encodes exactly the compatible global traces: each could have occurred as the actual execution. We implemented our monitoring approach in a tool which executes in parallel with the distributed system and takes as input the events generated from each scheduler and outputs the evaluated computation lattice. Our experimental results, omitted for space reasons, show that even for traces with thousands of events, the lattice size remains reasonable.

Future Work. The first extension of this work is to define how to use the computation lattice for efficiently evaluating properties at runtime. Moreover, we plan to decentralize the runtime monitors so that the satisfaction or violation of specifications can be detected by local monitors alone using decentralized monitoring techniques from [6, 14, 15] and decentralization/projection techniques for CBSs in [8, 22]. By distributing the monitors, we indeed decrease the load of monitoring process on a single entity. Another possible direction is to extend the proposed framework for timed components and timed specifications as presented in [33]. Finally, we plan to go beyond simple monitoring to allow components to react to errors by defining runtime enforcement [19] approaches for concurrent CBSs. For this, we plan extending our runtime enforcement approach [16] defined in the sequential setting to the multithreaded and distributed settings.

Acknowledgment. The authors thank the reviewers for their helpful comments.

The authors acknowledge the support from the H2020-ECSEL-2018-IA call – Grant Agreement number 826276 (CPS4EU), the European Union’s Horizon 2020 research and innovation programme - Grant Agreement number 956123 (FOCETA), from the French ANR project ANR-20-CE39-0009 (SEVERITAS), the Auvergne-Rhône-Alpes research project MOAP, and LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d’avenir.

References

1. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) *Lectures on Runtime Verification*. LNCS, vol. 10457, pp. 1–33. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1
2. Basu, A., et al.: Rigorous component-based system design using the BIP framework. *IEEE Softw.* **28**(3), 41–48 (2011)
3. Basu, A., Bidinger, P., Bozga, M., Sifakis, J.: Distributed semantics and implementation for systems with interaction and priority. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) *FORTE 2008*. LNCS, vol. 5048, pp. 116–133. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68855-6_8

4. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. *Formal Methods Syst. Des.* **48**(1–2), 46–93 (2016). <https://doi.org/10.1007/s10703-016-0253-8>
5. Bauer, A., Leucker, M., Schallhart, C.: Model-based runtime analysis of distributed reactive systems. In: *Proceedings of the Australian Software Engineering Conference (ASWEC 2006)*, pp. 243–252. IEEE (2006)
6. Bauer, A., Falcone, Y.: Decentralised LTL monitoring. In: Giannakopoulou, D., Méry, D. (eds.) *FM 2012. LNCS*, vol. 7436, pp. 85–100. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_10
7. Bensalem, S., Bozga, M., Quilbeuf, J., Sifakis, J.: Optimized distributed implementation of multiparty interactions with restriction. *Sci. Comput. Program.* **98**, 293–316 (2015)
8. Bistarelli, S., Martinelli, F., Matteucci, I., Santini, F.: A formal and run-time framework for the adaptation of local behaviours to match a global property. In: Kouchnarenko, O., Khosravi, R. (eds.) *FACS 2016. LNCS*, vol. 10231, pp. 134–152. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57666-4_9
9. Bliudze, S., Sifakis, J.: A notion of glue expressiveness for component-based systems. In: van Breugel, F., Chechik, M. (eds.) *CONCUR 2008. LNCS*, vol. 5201, pp. 508–522. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85361-9_39
10. Bonakdarpour, B., Bozga, M., Quilbeuf, J.: Automated distributed implementation of component-based models with priorities. In: Chakraborty, S., Jerraya, A., Baruah, S.K., Fischmeister, S. (eds.) *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, Part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, 9–14 October 2011*, pp. 59–68. ACM (2011)
11. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.: The FRACTAL component model and its support in Java. *Softw. Pract. Exp.* **36**(11–12), 1257–1284 (2006)
12. Colombo, C., Falcone, Y.: Organising LTL monitors over distributed systems with a global clock. *Formal Methods Syst. Des.* **49**(1–2), 109–158 (2016). <https://doi.org/10.1007/s10703-016-0251-x>
13. Dormoy, J., Kouchnarenko, O., Lanoix, A.: Using temporal logic for dynamic reconfigurations of components. In: Barbosa, L.S., Lumpe, M. (eds.) *FACS 2010. LNCS*, vol. 6921, pp. 200–217. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27269-1_12
14. El-Hokayem, A., Falcone, Y.: On the monitoring of decentralized specifications: semantics, properties, analysis, and simulation. *ACM Trans. Softw. Eng. Methodol.* **29**(1), 1:1–1:57 (2020)
15. Falcone, Y., Cornebize, T., Fernandez, J.-C.: Efficient and generalized decentralized monitoring of regular languages. In: Ábrahám, E., Palamidessi, C. (eds.) *FORTE 2014. LNCS*, vol. 8461, pp. 66–83. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43613-4_5
16. Falcone, Y., Jaber, M.: Fully automated runtime enforcement of component-based systems with formal and sound recovery. *Int. J. Softw. Tools Technol. Transfer* **19**(3), 341–365 (2016). <https://doi.org/10.1007/s10009-016-0413-6>
17. Falcone, Y., Jaber, M., Nguyen, T.-H., Bozga, M., Bensalem, S.: Runtime verification of component-based systems in the BIP framework with formally-proved sound and complete instrumentation. *Softw. Syst. Model.* **14**(1), 173–199 (2013). <https://doi.org/10.1007/s10270-013-0323-y>

18. Falcone, Y., Krstić, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transfer* **23**(2), 255–284 (2021). <https://doi.org/10.1007/s10009-021-00609-z>
19. Falcone, Y., Mariani, L., Rollet, A., Saha, S.: Runtime failure prevention and reaction. In: Bartocci, E., Falcone, Y. (eds.) *Lectures on Runtime Verification*. LNCS, vol. 10457, pp. 103–134. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_4
20. Fidge, C.J.: Timestamps in message-passing systems that preserve the partial ordering. *Austral. Comput. Sci. Commun.* **10**(1), 56–66 (1988)
21. Kouchnarenko, O., Weber, J.-F.: Adapting component-based systems at runtime via policies with temporal patterns. In: Fiadeiro, J.L., Liu, Z., Xue, J. (eds.) *FACS 2013*. LNCS, vol. 8348, pp. 234–253. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07602-7_15
22. Kouchnarenko, O., Weber, J.-F.: Decentralised evaluation of temporal patterns over component-based systems at runtime. In: Lanese, I., Madelaine, E. (eds.) *FACS 2014*. LNCS, vol. 8997, pp. 108–126. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-15317-9_7
23. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **21**(7), 558–565 (1978)
24. Lanese, I., Palacios, A., Vidal, G.: Causal-consistent replay reversible semantics for message passing concurrent programs. *Fundam. Informaticae* **178**(3), 229–266 (2021)
25. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. In: Rajamani, S.K., Walker, D. (eds.) *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, 15–17 January 2015*, pp. 221–232. ACM (2015)
26. Massart, T., Meuter, C.: Efficient online monitoring of LTL properties for asynchronous distributed systems. Technical report, Université Libre de Bruxelles (2006)
27. Mattern, F.: Virtual time and global states of distributed systems. *Parallel Distrib. Algorithms* **1**(23), 215–226 (1989)
28. Nazarpour, H., Falcone, Y., Bensalem, S., Bozga, M.: Concurrency-preserving and sound monitoring of multi-threaded component-based systems: theory, algorithms, implementation, and evaluation. *Formal Aspects Comput.* **29**(6), 951–986 (2017)
29. Nazarpour, H., Falcone, Y., Jaber, M., Bensalem, S., Bozga, M.: Monitoring distributed component-based systems. *ArXiv e-prints* (2017)
30. *Runtime Verification (2001–2021)*. <http://www.runtime-verification.org>
31. Sen, K., Vardhan, A., Agha, G., Rosu, G.: Efficient decentralized monitoring of safety in distributed systems. In: *Proceedings of the 26th International Conference on Software Engineering*, pp. 418–427. IEEE Computer Society (2004)
32. Tretmans, J.: A formal approach to conformance testing. In: *Protocol Test Systems, VI, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems*, pp. 257–276 (1993)
33. Triki, A., Combaz, J., Bensalem, S.: Optimized distributed implementation of timed component-based systems. In: *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pp. 30–35. IEEE (2015)

Author Index

- Baouya, Abdelhakim 27
Barbanera, Franco 82
Bensalem, Saddek 27, 153
Bozga, Marius 95, 153
- Chehida, Salim 27
- Dadeau, Frédéric 134
de Amorim, Arthur Azevedo 3
Di Gianantonio, Pietro 44
Di Luvre, Elena 63
- Falcone, Yliès 153
- Gianola, Alessandro 63
Groote, Jan Friso 115
Gros, Jean-Philippe 134
- Hijma, Pieter 115
- Iosif, Radu 95
- Jia, Limin 3
- Kouchnarenko, Olga 134
- Lanese, Ivan 82
- Martens, Jan 115
Miculan, Marino 44
- Nazarpour, Hosein 153
- Orlando, Simone 82
- Pășăreanu, Corina 3
Pasquale, Vairo Di 82
- Román, Mario 63
- Sabadini, Nicoletta 63
Sobociński, Paweł 63
Stolze, Claude 44
- Tuosto, Emilio 82
- van den Haak, Lars 115
- Wijs, Anton 115
- Zhang, Zichao 3