# EPTCS 387

Proceedings of the

# Combined 30th International Workshop on Expressiveness in Concurrency

## and 20th Workshop on Structural Operational Semantics

**Antwerp, Belgium, 18th September 2023**

Edited by: Claudio Antares Mezzina and Georgiana Caltais

Published: 14th September 2023
DOI: 10.4204/EPTCS.387
ISSN: 2075-2180
Open Publishing Association

# Table of Contents

# Preface

Georgiana Caltais
University of Twente, The Netherlands

Claudio Antares Mezzina
Università degli Studi di Urbino Carlo Bo, Urbino, Italy

This volume contains the proceedings of EXPRESS/SOS 2023, the Combined 30th International Workshop on Expressiveness in Concurrency (EXPRESS) and the 20th Workshop on Structural Operational Semantics (SOS). The first edition of EXPRESS/SOS was held in 2012, when the EXPRESS and SOS communities decided to organise an annual combined workshop bringing together researchers interested in the formal semantics of systems and programming concepts, and in the expressiveness of computational models. Since then, EXPRESS/SOS was held as one of the affiliated workshops of the International Conference on Concurrency Theory (CONCUR). Following this tradition, EXPRESS/SOS 2023 was held affiliated to CONCUR 2023, as part of CONFEST 2023, in Antwerp, Belgium. This year's edition marks two important anniversaries: **EXPRESS turns 30** and **SOS 20**. In celebration of this dual anniversary, and to offer an overview of the past and future of these workshops, we are delighted to present two contributions from distinguished members of both communities in these proceedings:

- *EXPRESSing Session Types*, by Ilaria Castellani, Ornela Dardha, Luca Padovani and Davide Sangiorgi;
- *The Way We Were: Structural Operational Semantics Research in Perspective*, by Luca Aceto, Pierluigi Crescenzi, Anna Ingólfsdóttir and Mohammad Reza Mousavi.

The topics of interest for the EXPRESS/SOS workshop include (but are not limited to):

- expressiveness and rigorous comparisons between models of computation;
- expressiveness and rigorous comparisons between programming languages and models;
- logics for concurrency; analysis techniques for concurrent systems;
- comparisons between structural operational semantics and other formal semantic approaches;
- applications and case studies of structural operational semantics;
- software tools that automate, or are based on, structural operational semantics.

This volume contains revised versions of the 7 full papers and one abstract, selected by the Program Committee, as well as the following two invited papers, related to the topics presented by our invited speakers:

- Formalizing Real World Programming Languages with Skeletal Semantics , by Alan Schmitt (INRIA, France);
- Timed Actors and their Formal Verification, by Marjan Sirjani (Malardalen University, Sweden).

We would like to thank the authors of the submitted papers, the invited speakers, the members of the program committee, and their subreviewers for their contribution to both the meeting and this volume. We also thank the CONCUR 2023 and the CONFEST 2023 organizing committees for hosting the workshop. Finally, we would like to thank our EPTCS editor Rob van Glabbeek for publishing these proceedings

G. Caltais and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2023).
EPTCS 387, 2023, pp. ii–iii, doi:10.4204/EPTCS.387.0

and his help during the preparation.

Georgiana Caltais and Claudio Antares Mezzina,
August 2023

**Program Committee**

- Valentina Castiglioni, Reykjavik University, Iceland
- Matteo Cimini, University of Massachusetts Lowell, US
- Cinzia Di Giusto, Université Côte d'Azur, France / CNRS, France
- Wan Fokkink, Vrije Universiteit Amsterdam, The Netherlands
- Sergey Goncharov, FAU Erlangen-Nürnberg, Germany
- Tobias Kappé, Open University of the Netherlands and ILLC, University of Amsterdam, The Netherlands
- Vasileios Koutavas, Trinity College Dublin, Ireland
- Bas Luttik, Eindhoven University of Technology, The Netherlands
- Hernán Melgratti, Universidad de Buenos Aires, Argentina
- Mohammadreza Mousavi, King's College London, UK
- Jorge A. Pérez, University of Groningen, The Netherlands
- G. Michele Pinna, Università di Cagliari, Italy
- Max Tschaikowski, Aalborg University, Denmark

**Additional Reviewers**

- Andrea Esposito, Università degli Studi di Urbino Carlo Bo, Urbino, Italy

# Timed Actors and Their Formal Verification

Marjan Sirjani
Mälardalen University
Västerås, Sweden
marjan.sirjani@mdu.se

Ehsan Khamespanah
University of Tehran
Tehran, Iran
e.khamespanah@ut.ac.ir

In this paper we review the actor-based language, Timed Rebeca, with a focus on its formal semantics and formal verification techniques. Timed Rebeca can be used to model systems consisting of encapsulated components which communicate by asynchronous message passing. Messages are put in the message buffer of the receiver actor and can be seen as events. Components react to these messages/events and execute the corresponding message/event handler. Real-time features, like computation delay, network delay and periodic behavior, can be modeled in the language. We explain how both Floating-Time Transition System (FTTS) and common Timed Transition System (TTS) can be used as the semantics of such models and the basis for model checking. We use FTTS when we are interested in event-based properties, and it helps in state space reduction. For checking the properties based on the value of variables at certain point in time, we use the TTS semantics. The model checking toolset supports schedulability analysis, deadlock and queue-overflow check, and assertion based verification of Timed Rebeca models. TCTL model checking based on TTS is also possible but is not integrated in the tool.

## 1 Introduction

Actors are introduced for modeling and implementation of distributed systems [7, 3]. Timed Actors allow us to introduce timing constraints, and progress of time, and are most useful for modeling time-sensitive systems. Timed Rebeca is one of the first timed actor languages with model checking support [10]. Timed Rebeca restricts the modeller to a pure asynchronous actor-based paradigm, where the structure of the model can represent the service oriented architecture, while the computational model matches the network infrastructure [1]. In a different context, it may represent components of cyber-physical systems, where components are triggered by events put in their input buffers, or by time events [14]. Timed Rebeca is equipped with analysis techniques based on the standard semantics of timed systems, and also an innovative event-based semantics that is tailored for timed actor models [13].

Timed Rebeca is an extension of the Reactive Object Language, Rebeca [16]. It is reviewed and compared to a few other actor languages in a survey published in ACM Computing Surveys in 2017 [4]. The very first ideas of Rebeca and its compositional verification is presented at AVoCS workshop in 2001 [15]. Timed Rebeca, different formal semantics of it, and the model checking support are presented in multiple papers. Here we present an overall view and insight into different semantics and use a simple example to show the differences visually.

## 2 Timed Rebeca

A Timed Rebeca model mainly consists of a number of *reactive class* definitions. These reactive classes define the behavior of the classes of the actors in the model. The model also has a `main` block that defines the instances of the actor classes.

G. Caltais and C. A. Mezzina (Eds): Combined Workshop on
Expressiveness in Concurrency and Structural Operational Semantics
(EXPRESS/SOS 2023).
EPTCS 387, 2023, pp. 1–7, doi:10.4204/EPTCS.387.1

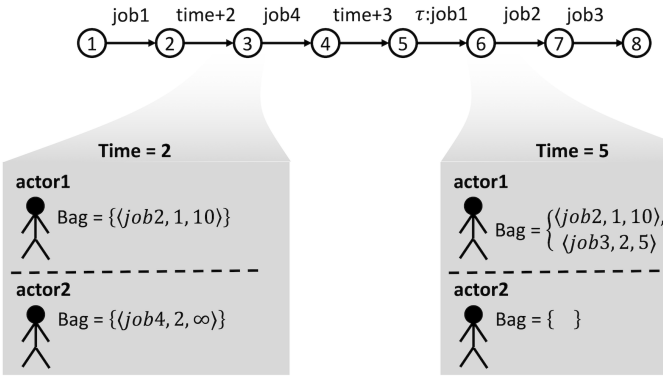
We use a simple Timed Rebeca model as an example to explain the language features. In this example we consider two different actors. The first actor is able to handle three different tasks, named as *job1*, *job2*, and *job3*. The second actor can only handle one task, named as *job4*. The Timed Rebeca model of this example is shown in Listing 1. there are two classes of actors: `Actor1` (lines 1-15) and `Actor2` (lines 17-27). The `main` block in lines 29-32 defines one instance of each class. Each reactive class has a number of *state variables*, representing the local state of the actors. They may contain variables of basic data types, including booleans, integers, arrays, or references to other actors. To make the example model simple, none of the reactive classes of Listing 1 has any state variables. Each class can have a *constructor*, which is used to initialize the created instances of the class by initializing the state variables, and start up running of the model by sending messages to itself or other actors.

In the Timed Rebeca model of Listing 1, in the constructor of `Actor1` (line 3), the actor sends itself a `job1` message. Each reactive class accepts a number of message types which are handled using *message servers* (`msgsrv`). `Actor1` has three message servers, `job1` (lines 5-8), `job2` (lines 9-11), and `job3` (lines 12-14). Serving a message of type `job1` results in sending `job2` message to self which is put in the message buffer of itself only after passing 1 unit of time (modeled by using the `after` construct). The `deadline` construct denotes the deadline of the message to be handled, if at the time of handling the event the deadline is passed the model checking tool notifies that. Then, there is a `delay` statement which models progress of time for 5 units of time, this can be used to model a computation delay. In the definition of the message servers, well-known program control structures can be used, including *if-else* conditional statements, *for* and *while* loops, the definition of local variables, and assignments using usual arithmetic, logic, and comparative operators.

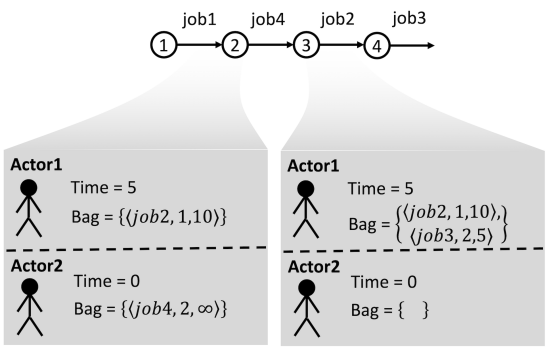
Listing 1: A simple Timed Rebeca model with two actors.

```
1  reactiveclass Actor1(3) {
2    Actor1() {
3      self.job1();
4    }
5    msgsrv job1() {
6      self.job2() after(1) deadline(10);
7      delay(5);
8    }
9    msgsrv job2() {
10
11   }
12   msgsrv job3() {
13     self.job3() after(1);
14   }
15 }
16
17 reactiveclass Actor2(3) {
18   knownrebecs {
19     Actor1 a1;
20   }
21   Actor2() {
22     self.job4() after(2);
23   }
24   msgsrv job4() {
25     a1.job3() after(2) deadline(5);
26   }
27 }
28
29 main {
30   Actor1 actor1():();
31   Actor2 actor2(actor1):();
32 }
```

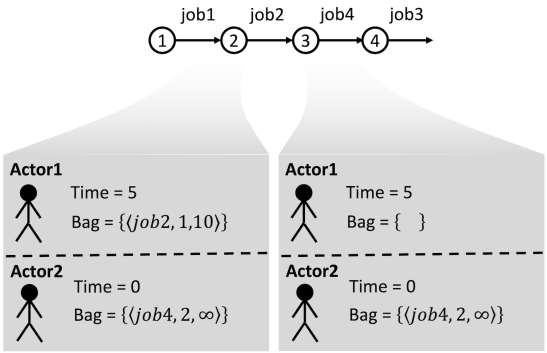
In Timed Rebeca models, we assume that actors have local clocks which are synchronized throughout the model. Each message is tagged with a time stamp (called a time tag). We use a `delay(t)` statement to model the computation delay, and we use `after(t)` in combination with a `send` message statement to model a network delay, or model a periodic event. When we use `after(t)` in a `send` message statement it means that the time tag of the message when it is put in the queue of the receiver is the value of the local clock of the sender plus the value of `t`. The progress of time is forced by the `delay` statement.




(a) TTS of the Timed Rebeca model of Listing 1




(b) FTTS of the Timed Rebeca model of Listing 1. Each actor has its own local clock represented by Time. The value of local clock is considered in choosing the next transition.




(c) Relaxed-FTTS of the Timed Rebeca model of Listing 1. Each actor has its own local clock represented by Time. The message with the lowest time tag is chosen and the execution of its message server is the label of the next transition.

Figure 1: Comparing TTS, FTTS, and the relaxed form of FTTS for the Timed Rebeca model of Listing 1.

We assume that the local clock of each actor is zero when the model starts execution, the local clock is increased by value of `t` if there is a `delay(t)` statement. A `send` statement with an `after` does not cause an increase in the local time necessarily. The local time of the receiver actor is set to the time tag of the message when the actor picks the message, unless it is already greater than that. The latter situation means that the message sits in the queue while the actor is busy executing another message, in this case the `after` construct does not cause progress of time. The progress of time happens in the case that the time tag of the message is greater than the local time of the receiver actor, in this case the local time will be pushed forward. In Timed Rebeca, messages are executed atomically and are not preempted.

## 3 Different Semantics of Timed Rebeca

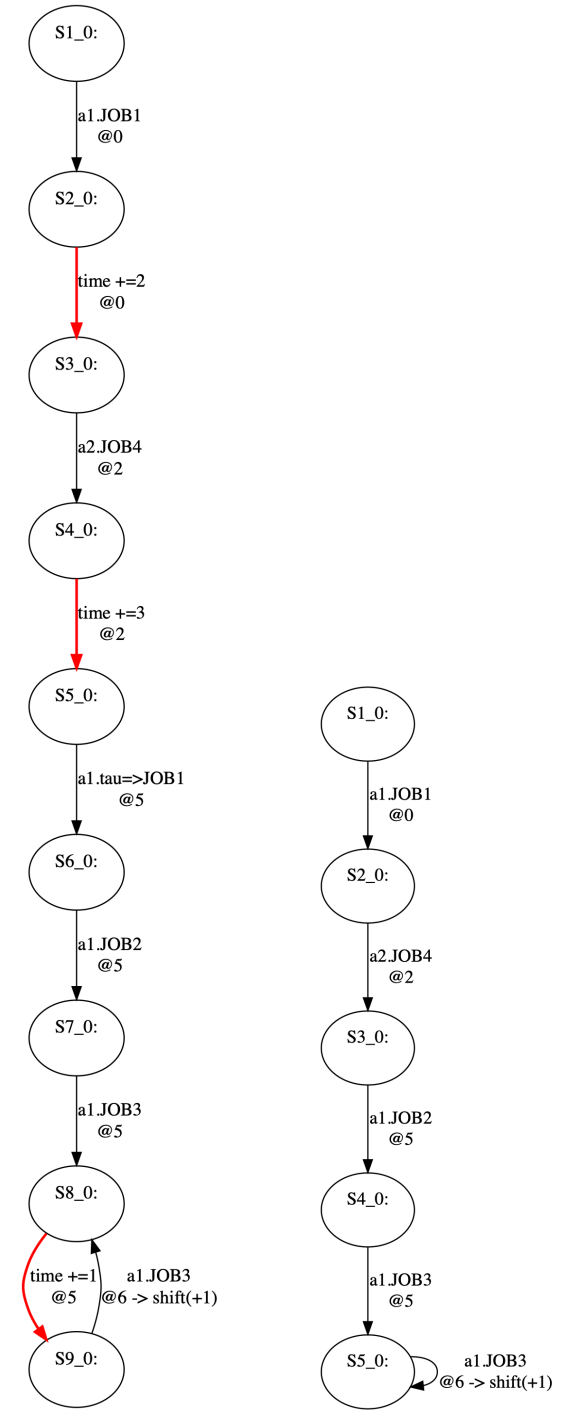
We first introduced an event-based semantics for Timed Rebeca and used McErlang for simulation of Timed Rebeca models in [1, 12]. In this semantics we focused on the object-based features of actors, encapsulation and information hiding, and decided on a coarse-grained semantics where serving a message (or handing a request or signal) are the only observable behavior of actors. We considered taking a message from top of the message queue and executing it as an observable action, and we called it an

event. Note that by a message queue in Timed Rebeca, we mean a bag of messages where each message has a time tag of when the message is put in the buffer. Here, by "top of the message queue of an actor", we mean the message with the least time tag in the bag of messages targeted to that actor. In defining the formal semantics of Timed Rebeca as a labeled transition system, we only have one type of label on the transitions, *events*, which are taking messages and executing them. In [11], and its extended version [10], we introduced this event-based semantics of Timed Rebeca as Floating Time Transition System (FTTS) and compared it with the time semantics that is generally used for timed models (for example for Timed CCS [2]) where the transitions can be of the type of an event, progress of time, and a silent action.

Although we consider FTTS as the original and most fit semantics for Timed Actors, it may also be seen as a reduction technique in model checking. FTTS can give a significant reduction in state space compared to the standard Timed Transition System. In [10] we proved that there is an action-based weak bisimulation relation between FTTS and TTS of Timed Rebeca. Note that the focus here is on the labels on the transitions not on the values of variables in the states.

The semantics presented in [12], is a relaxed form of FTTS in [10] where in choosing the next step in a state we have a simpler policy. The SOS rules of FTTS and the relaxed version are presented in [10] and [12], respectively. In each state, the SOS rule for the scheduler chooses the next message in the bags of actors to be executed. In the relaxed form of FTTS, the scheduler simply chooses the message with the least time tag (targeted to any actor). In FTTS, the schedular considers the local clock of each actor as well. For each actor, the maximum between the local clock and the lowest time tag of the messages in the message bag of the actor is computed. Then among all the actors, the scheduler chooses the actor with the least of these amounts. The message on the top of the queue of this chosen actor will be executed next. Comparing to the standard TTS, the relaxed form of FTTS preserves the order of execution of messages of all actors if we consider the time tags of messages for ordering. The intuitive reason is that in Timed Rebeca we consider a FIFO policy for scheduling the messages in the message buffer, when we choose the message with the lowest time tag to be executed, it is guaranteed that from that point on, there will be no messages with a smaller time tag added to the message buffer (of any actor). So, the FIFO policy for serving messages can be correctly respected. The subtle point here is that the actor $a$ with the lowest time tag message $m$ may be busy when message $m$ is sitting in its message buffer, in the meanwhile other messages from other actors may get the chance to be executed and send messages to actor $a$. Of course, the time tag of those messages will be greater than the time tag of message $m$, but still we are losing the "correct" content of the message buffer of $a$ at some snapshots in time. By this observation, we moved to the FTTS semantics in [10] where at any point in time, we have the correct content of the message buffer. Using this semantics we may choose to use other scheduling policies for messages (events) in the buffer, for example the *earliest deadline first* policy.

In Figure 1, we show parts of the the state transition system for Rebeca model in Listing 1. In this figure, we see how in TTS we may have three types of labels on the transitions, an event, time progress and $\tau$ (silent) transitions. In FTTS, in state 2 in Figure 1.b, the scheduler chooses the message `<job4,2,∞>` while in the relaxed form of FTTS, in Figure 1.c, the scheduler chooses the message `<job2,1,10>`. The reason is that although the message with the lowest time tag is `<job2,1,10>`, with the time tag 1, the maximum between 1 and the value for the local clock of `Actor1` is 5. The maximum between 2 (the time tag for message `<job4,2,∞>` ) and the value for the local clock of `Actor2` (which is zero in this state) is 2.




(a) The state space for TTS-based semantics

(b) The state space for FTTS-based semantics

Figure 2: The state space of the Rebeca model of Listing 1, generated by Afra using TTS and FTTS semantics

## 4 Model Checking Timed Rebeca Models

The verification algorithms of TTS with dense time are generally PSPACE-complete as stated in [6]. In the existing model checking tools commonly the properties are limited to a subset of TCTL properties without nested timed quantifiers. For this subset efficient algorthms are developed. In the case of Timed Rebeca, we use *discrete time*, and hence TTS can be verified efficiently in polynomial time against TCTL properties. Discrete time is the time model in which passage of time is modeled by natural numbers. We developed a model checking tool and a reduction technique for Timed Rebeca models based on TTS semantics against TCTL properties [8]. This toolset is not integrated in the Afra IDE [9].

We also developed a tool for the model checking of Timed Rebeca models based on both TTS and FTTS semantics, which is integrated in Afra. The current implementation of the model checking toolset supports schedulability analysis, and checking for deadlock-freedom, queue-overflow freedom, and assertion based verification of Timed Rebeca models. Note that in FTTS, in each state actors may have different local clocks, so, writing meaningful assertion needs special care. Assertions on state variables of one actor are not problematic. The Timed Rebeca code of the case studies and the model checking toolset are accessible from Rebeca homepage [5].

Figure 2 shows the state space generated automatically by the model checking tool, Afra, for the Timed Rebeca model in Listing 1 based on the two semantics, TTS and FTTS. It is shown that the order of events are preserved while time progress and $\tau$ transitions are hidden. In state $S9\_0$ in Figure 2.a, and state $S5\_0$ in Figure 2.b, you see how the transition system becomes bounded using a shift operation on the time. The shift keyword means that for example by the event `a1.JOB3`, we go back to state $S8\_0$ (or $S5\_0$), where all the values of state variables, local variables and messages in the message buffers stay the same but the value of parameters related to time (including time tag of all messages and local clock value) change and have a shift by the same value.

## References

[1] Luca Aceto, Matteo Cimini, Anna Ingólfsdóttir, Arni Hermann Reynisson, Steinar Hugi Sigurdarson & Marjan Sirjani (2011): *Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca*. In Mohammad Reza Mousavi & António Ravara, editors: *Proceedings 10th International Workshop on the Foundations of Coordination Languages and Software Architectures, FOCLASA 2011, Aachen, Germany, 10th September, 2011*, *EPTCS* 58, pp. 1–19, doi:10.4204/EPTCS.58.1.

[2] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen & Jiri Srba (2007): *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, USA, doi:10.1017/CBO9780511814105.

[3] Gul Agha (1986): *Actors: a model of concurrent computation in distributed systems*. MIT press, doi:10.7551/mitpress/1086.001.0001.

[4] Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes & Albert Mingkun Yang (2017): *A Survey of Active Object Languages*. *ACM Comput. Surv.* 50(5), pp. 76:1–76:39, doi:10.1145/3122848.

[5] Rebeca Research Group: *Afra toolset homepage*. Available at `http://rebeca-lang.org/alltools/Afra`.

[6] Thomas A. Henzinger, Zohar Manna & Amir Pnueli (1991): *Timed Transition Systems*. In J. W. de Bakker, Cornelis Huizing, Willem P. de Roever & Grzegorz Rozenberg, editors: *Real-Time: Theory in Practice, REX Workshop, Mook, The Netherlands, June 3-7, 1991, Proceedings*, *Lecture Notes in Computer Science* 600, Springer, pp. 226–251, doi:10.1007/BFb0031995.

[7] Carl Hewitt (1977): *Viewing control structures as patterns of passing messages*. Artificial intelligence 8(3), pp. 323–364, doi:10.1016/0004-3702(77)90033-9.

[8] Ehsan Khamespanah, Ramtin Khosravi & Marjan Sirjani (2018): *An efficient TCTL model checking algorithm and a reduction technique for verification of timed actor models*. Sci. Comput. Program. 153, pp. 1–29, doi:10.1016/j.scico.2017.11.004.

[9] Ehsan Khamespanah, Ramtin Khosravi & Marjan Sirjani (2023): *Afra: An Eclipse-Based Tool with Extensible Architecture for Modeling and Model Checking of Rebeca Family Models*. In Hossein Hojjat & Mieke Massink, editors: Fundamentals of Software Engineering - 11th International Conference, FSEN 2023, May 4-5, 2023, Lecture Notes in Computer Science, Springer, doi:10.1007/978-3-031-42441-0_6.

[10] Ehsan Khamespanah, Marjan Sirjani, Zeynab Sabahi-Kaviani, Ramtin Khosravi & Mohammad-Javad Izadi (2015): *Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system*. Sci. Comput. Program. 98, pp. 184–204, doi:10.1016/j.scico.2014.07.005.

[11] Ehsan Khamespanah, Marjan Sirjani, Mahesh Viswanathan & Ramtin Khosravi (2015): *Floating Time Transition System: More Efficient Analysis of Timed Actors*. In Christiano Braga & Peter Csaba Ölveczky, editors: Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers, Lecture Notes in Computer Science 9539, Springer, pp. 237–255, doi:10.1007/978-3-319-28934-2_13.

[12] Arni Hermann Reynisson, Marjan Sirjani, Luca Aceto, Matteo Cimini, Ali Jafari, Anna Ingólfsdóttir & Steinar Hugi Sigurdarson (2014): *Modelling and simulation of asynchronous real-time systems using Timed Rebeca*. Sci. Comput. Program. 89, pp. 41–68, doi:10.1016/j.scico.2014.01.008.

[13] Marjan Sirjani & Ehsan Khamespanah (2016): *On Time Actors*. In Erika Ábrahám, Marcello M. Bonsangue & Einar Broch Johnsen, editors: Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday, Lecture Notes in Computer Science 9660, Springer, pp. 373–392, doi:10.1007/978-3-319-30734-3_25.

[14] Marjan Sirjani, Edward A. Lee & Ehsan Khamespanah (2020): *Verification of Cyberphysical Systems*. Mathematics 8(7), doi:10.3390/math8071068.

[15] Marjan Sirjani, Ali Movaghar & Mohammadreza Mousavi (2001): *Compositional verification of an object-based reactive system*. In: Workshop on Automated Verification of Critical Systems AVoCS 2001.

[16] Marjan Sirjani, Ali Movaghar, Amin Shali & Frank S. de Boer (2004): *Modeling and Verification of Reactive Systems using Rebeca*. Fundam. Informaticae 63(4), pp. 385–410. Available at `http://content.iospress.com/articles/fundamenta-informaticae/fi63-4-05`.

# EXPRESSing Session Types

Ilaria Castellani
INRIA, Université Côte d'Azur

Ornela Dardha
University of Glasgow

Luca Padovani
University of Camerino

Davide Sangiorgi
University of Bologna, INRIA

To celebrate the 30th edition of EXPRESS and the 20th edition of SOS we overview how session types can be expressed in a type theory for the standard $\pi$-calculus by means of a suitable encoding. The encoding allows one to reuse results about the $\pi$-calculus in the context of session-based communications, thus deepening the understanding of sessions and reducing redundancies in their theoretical foundations. Perhaps surprisingly, the encoding has practical implications as well, by enabling refined forms of deadlock analysis as well as allowing session type inference by means of a conventional type inference algorithm.

## 1 Origins of EXPRESS: some personal memories

This year marks an important milestone in the history of the EXPRESS/SOS workshop series. Before joining their destinies in 2012, the two workshops EXPRESS and SOS had been running on their own since 1994 and 2004, respectively. Hence, the EXPRESS/SOS'23 workshop in Antwerp will constitute the 30th edition of EXPRESS and the 20th edition of SOS.

Two of us (Ilaria Castellani and Davide Sangiorgi) were personally involved in the very first edition of EXPRESS in 1998, and indeed, they may be said to have carried the workshop to the baptismal font, together with Robert de Simone and Catuscia Palamidessi. Let us recall some facts and personal memories. The EXPRESS workshops were originally held as meetings of the European project EXPRESS, a Network of Excellence within the Human Capital and Mobility programme, dedicated to expressiveness issues in Concurrency Theory. This NoE, which lasted from January 1994 till December 1997, gathered researchers from several European countries and was particularly fruitful in supporting young researchers' mobility across different sites. The first three workshops of the NoE were held in Amsterdam (1994), Tarquinia (1995), and Dagstuhl (1996). The fourth and final workshop was held in Santa Margherita Ligure (1997). It was co-chaired by Catuscia Palamidessi and Joachim Parrow, and stood out as a distinctive event, open to external participants and organised as a conference with a call for papers. A few months after this workshop, in the first half of 1998, the co-chairs of the forthcoming CONCUR'98 conference in Nice, Robert de Simone and Davide Sangiorgi, were wondering about endowing CONCUR with a satellite event (such events were still unusual at the time) in order to enhance its attractiveness. Moreover, Davide was sharing offices with Ilaria, who had been the NoE responsible for the site of Sophia Antipolis and was also part of the organising committee of CONCUR'98. It was so, during informal discussions, that the idea of launching EXPRESS as a stand-alone event affiliated with CONCUR was conceived, in order to preserve the heritage of the NoE and give it a continuation. Thus the first edition of EXPRESS, jointly chaired by Catuscia and Ilaria, took place in Nice in 1998, as the first and unique satellite event of CONCUR. However, EXPRESS did not remain a lonely satellite for too long, as other workshops were to join the orbit of CONCUR in the following years (INFINITY, YR-CONCUR, SecCo, TRENDS, ...), including SOS in 2004. The workshop EXPRESS'98 turned out

G. Caltais and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2023).
EPTCS 387, 2023, pp. 8–25, doi:10.4204/EPTCS.387.2

to be successful and very well attended. Since then, EXPRESS has been treading its path as a regular satellite workshop of CONCUR, with a new pair of co-chairs every year, each co-chair serving two editions in a row. The workshop, which is traditionally held on the Monday preceding CONCUR, has always attracted good quality submissions and has maintained a faithful audience over the years.

Coincidentally, this double anniversary of EXPRESS/SOS falls in the 30th anniversary of Kohei Honda's first paper on session types [26]. For this reason, we propose an overview of a particular expressiveness issue, namely the addition of session types to process calculi for mobility such as the $\pi$-calculus.

## 2 Session types and their expressiveness: introduction

Expressiveness is a key topic in the design and implementation of programming languages and models. The issue is particularly relevant in the case of formalisms for parallel and distributed systems, due to the breadth and variety of constructs that have been proposed.

Most importantly, the study of expressiveness has practical applications. If the behaviours that can be programmed by means of a certain formalism $L_1$ can also be programmed using another formalism $L_2$, then methods and concepts developed for the latter language (e.g., reasoning and implementation techniques) may be transferred onto the former one that, in turn, may be more convenient to use from a programming viewpoint. An important instance is the case when $L_2$ is, syntactically, a subset of $L_1$. Indeed the quest for a "minimal" formalism is central in the work on expressiveness.

This paper is an overview of a particular expressiveness issue, namely the addition of session types onto calculi for mobility such as the $\pi$-calculus. We will review the encoding of binary session types onto the standard $\pi$-calculus [14, 15], based on an observation of Kobayashi [33]. The key idea of the encoding is to represent a sequence of communications within a session as a chain of communications on linear channels (channels that are meant to be used exactly once) through the use of explicit continuations, a technique that resembles the modelling of communication patterns in the actor model [25]. We discuss extensions of the encoding to subtyping, polymorphism and higher-order communication as well as multiparty session types. Finally, we review two applications of the encoding to the problems of deadlock analysis and of session type inference.

*Session types*, initially proposed in [26, 51, 27], describe *sessions*, i.e., interaction protocols in distributed systems. While originally designed for process calculi, they have later been integrated also in other paradigms, including (multi-threaded) functional programming [54, 44, 37, 40, 20, 35], component-based systems [52], object-oriented languages [18, 19, 7], languages for Web Services and Contracts [9, 38]. They have also been studied in logical-based type systems [5, 55, 6, 13, 36].

Session types allow one to describe the sequences of input and output operations that the participants of a session are supposed to follow, explicitly indicating the types of messages being transmitted. This structured *sequentiality* of operations makes session types suitable to model protocols. Central (type and term) constructs in session types are also branch and select, the former being the offering of a set of alternatives and the latter being the selection of one of the possible options at hand.

Session types were first introduced in a variant of the $\pi$-calculus to describe binary interactions. Subsequently, they have been extended to *multiparty sessions* [28], where several participants interact with each other. In the rest of this paper, we will focus on *binary session types*.

Session types guarantee privacy and communication safety within a session. Privacy means that session channels are known and used only by the participants involved in the session. Communication safety means that interaction within a session will proceed without mismatches of direction and of message type. To achieve this, a session channel is split into two endpoints, each of which is owned by one

of the participants. These endpoints are used according to dual behaviours (and thus have dual types), namely one participant sends what the other one is expecting to receive and vice versa. Indeed, *duality* is a key concept in the theory of session types.

To better understand session types and the notion of duality, let us consider a simple example: the *equality test*. A *server* and a *client* communicate over a session channel. The endpoints $x$ and $y$ of the session channel are owned by the server and the client, respectively and exclusively, and must have dual types. To guarantee duality of types, static checks are performed by the type system.

If the type of the server endpoint $x$ is

$$S \triangleq ?\texttt{Int}.?\texttt{Int}.!\texttt{Bool}.\texttt{end}$$

— meaning that the process owning the channel endpoint $x$ receives (?) an integer value followed by another integer value and then sends (!) back a boolean value corresponding to the equality test of the integers received — then the type of the client endpoint $y$ should be

$$\overline{S} \triangleq !\texttt{Int}.!\texttt{Int}.?\texttt{Bool}.\texttt{end}$$

— meaning that the process owning the channel endpoint $y$ sends an integer value followed by another integer value and then waits to receive back a boolean value — which is exactly the dual type.

There is a precise moment at which a session between two participants is established. It is the *connection* phase, when a fresh (private) session channel is created and its endpoints are bound to each communicating process. The connection is also the moment when duality, hence mutual compliance of two session types, is verified. In order to establish a connection, primitives like `accept/request` or $(\nu xy)$, are added to the syntax of terms [51, 27, 53].

When session types and session terms are added to the syntax of standard $\pi$-calculus types and terms, respectively, the syntax of types (and, as a consequence, of type environments) usually needs to be split into two separate syntactic categories, one for session types and the other for standard $\pi$-calculus types [51, 27, 56, 22]. Common typing features, like subtyping, polymorphism, recursion have then to be added to both syntactic categories. Also the syntax of processes will contain both standard $\pi$-calculus process constructs and session process constructs (for example, the constructs mentioned above to create session channels). These syntactic redundancies bring in redundancies also in the theory, and can make the proofs of properties of the language heavy. Moreover, if a new type construct is added, the corresponding properties must be checked both on standard $\pi$-types and on session types. By "standard type systems" we mean type systems originally studied in depth for sequential languages such as the $\lambda$-calculus and then transplanted onto the $\pi$-calculus as types for channel names (rather than types for terms as in the $\lambda$-calculus); they include, for instance, constructs for products, records, variants, polymorphism, linearity, capabilities, and so on.

A further motivation for investigating the expressiveness of the $\pi$-calculus with or without session types is the similarity between session constructs and standard $\pi$-calculus constructs. Consider the type $S = ?\texttt{Int}.?\texttt{Int}.!\texttt{Bool}.\texttt{end}$. This type is assigned to a session channel endpoint and it describes a structured sequence of inputs and outputs by specifying the type of messages that the channel can transmit. This way of proceeding reminds us of the *linearised* channels [34], which are channels used multiple times for communication but only in a sequential manner. Linearised types can, in turn, be encoded into linear types—i.e., channel types used *exactly once* [34]. Similarly, there are analogies between the branch and select constructs of session types and the *variant* types [45, 46] of standard $\pi$-calculus types, as well as between the duality of session types, in which the behaviour of a session channel is split into

| | | | | | |
|---|---|---|---|---|---|
| $T ::=$ | $S$ | (session type) | $S ::=$ | $\texttt{end}$ | (termination) |
| | $\sharp T$ | (channel type) | | $!T.S$ | (send) |
| | $\texttt{Unit}$ | (unit type) | | $?T.S$ | (receive) |
| | $\ldots$ | (other types) | | $\oplus\{l_i : S_i\}_{i\in I}$ | (select) |
| | | | | $\&\{l_i : S_i\}_{i\in I}$ | (branch) |

| | | | | |
|---|---|---|---|---|
| $P,Q ::=$ | $x!\langle v\rangle.P$ | (output) | $\mathbf{0}$ | (inaction) |
| | $x?(y).P$ | (input) | $P \mid Q$ | (composition) |
| | $x \triangleleft l_j.P$ | (selection) | $(\nu xy)P$ | (session restriction) |
| | $x \triangleright \{l_i : P_i\}_{i\in I}$ | (branching) | $(\nu x)P$ | (channel restriction) |
| $v ::=$ | $x$ | (name) | $\star$ | (unit value) |

$$
\begin{array}{rl}
(\text{R- StndCom}) & x!\langle v\rangle.P \mid x?(z).Q \to P \mid Q[v/z] \\
(\text{R-Com}) & (\nu xy)(x!\langle v\rangle.P \mid y?(z).Q) \to (\nu xy)(P \mid Q[v/z]) \\
(\text{R-Case}) & (\nu xy)(x \triangleleft l_j.P \mid y \triangleright \{l_i : P_i\}_{i\in I}) \to (\nu xy)(P \mid P_j) \quad j \in I \\
(\text{R-StndRes}) & P \to Q \Longrightarrow (\nu x)P \to (\nu x)Q \\
(\text{R-Res}) & P \to Q \Longrightarrow (\nu xy)P \to (\nu xy)Q \\
(\text{R-Par}) & P \to Q \Longrightarrow P \mid R \to Q \mid R \\
(\text{R-Struct}) & P \equiv P',\ P \to Q,\ Q' \equiv Q \Longrightarrow P' \to Q'
\end{array}
$$

Figure 1: Syntax and reduction semantics of the session $\pi$-calculus

two endpoints, and the *capability types* of the standard $\pi$-calculus, that allow one to separate the input and output usages of channels.

In this paper we follow the encoding of binary session types into linear $\pi$-types from [14, 15], then discuss some extensions and applications. The encoding was first suggested by Kobayashi [33], as a proof-of-concept without however formally studying it. Later, Demangeon and Honda [17] proposed an encoding of session types into $\pi$-types with the aim of studying the subtyping relation, and proving properties such as soundness of the encoding with respect to typing and full abstraction.

**Structure of the paper.** The rest of the paper is organised as follows. In Section 3 we introduce the necessary background about the session $\pi$-calculus and the linear $\pi$-calculus. In Section 4 we recall the encoding from the session $\pi$-calculus into the linear $\pi$-calculus, as well as its correctness result. In Section 5 and Section 6 we discuss respectively some extensions and some applications of the encoding.

## 3 Background: $\pi$-calculus and session types

In this section, we recall the syntax and semantics of our two calculi of interest: the session $\pi$-calculus and the standard typed $\pi$-calculus. We also introduce the notion of duality for session types.

$$
\begin{array}{rlll|lll}
t ::= & \ell_{\mathtt{o}}[\widetilde{t}] & \text{(linear output)} & & \sharp[\widetilde{t}] & \text{(connection)} \\
 & \ell_{\mathtt{i}}[\widetilde{t}] & \text{(linear input)} & & \langle l_i \_ t_i \rangle_{i \in I} & \text{(variant type)} \\
 & \ell_{\sharp}[\widetilde{t}] & \text{(linear connection)} & & \mathtt{Unit} & \text{(unit type)} \\
 & \emptyset[] & \text{(no capability)} & & \ldots & \text{(other types)}
\end{array}
$$

$$
\begin{array}{rlll|lll}
P,Q ::= & x!\langle \widetilde{v} \rangle.P & \text{(output)} & & \mathbf{0} & \text{(inaction)} \\
 & x?(\widetilde{y}).P & \text{(input)} & & P \mid Q & \text{(composition)} \\
 & (\nu x)P & \text{(restriction)} & & \mathbf{case}\ v\ \mathbf{of}\ \{l_i \_ (x_i) \triangleright P_i\}_{i \in I} & \text{(case)} \\
 & & & & & \\
v ::= & x & \text{(name)} & & \star & \text{(unit value)} \\
 & l \_ v & \text{(variant value)} & & &
\end{array}
$$

$$
\begin{array}{rl}
(\text{R}\pi\text{-Com}) & x!\langle \tilde{v} \rangle.P \mid x?(\tilde{z}).Q \to P \mid Q[\tilde{v}/\tilde{z}] \\
(\text{R}\pi\text{-Case}) & \mathbf{case}\ l_j \_ v\ \mathbf{of}\ \{l_i \_ (x_i) \triangleright P_i\}_{i \in I} \to P_j[v/x_j] \quad j \in I \\
(\text{R}\pi\text{-Res}) & P \to Q \Longrightarrow (\nu x)P \to (\nu x)Q \\
(\text{R}\pi\text{-Par}) & P \to Q \Longrightarrow P \mid R \to Q \mid R \\
(\text{R}\pi\text{-Struct}) & P \equiv P',\ P \to Q,\ Q' \equiv Q \Longrightarrow P' \to Q'
\end{array}
$$

Figure 2: Syntax and reduction rules of the standard typed $\pi$-calculus

**Session types and terms.** The syntax for session types and session $\pi$-calculus terms is reported in Figure 1, together with the rules for the reduction semantics, in which $\equiv$ is the usual *structural congruence* relation, allowing one to rearrange parallel compositions and the scope of restrictions and to remove useless restrictions. We refer to, e.g., [53, 22] for the rules for typing. Session types range over $S$ and types range over $T$; the latter include session types, standard channel types denoted by $\sharp T$, data types, such as `Unit` and any other type construct needed for mainstream programming.

Session types are: `end`, the type of a terminated channel; $?T.S$ and $!T.S$ (used in the equality test example given in the introduction) indicating, respectively, the receive and send of a value of type $T$, with continuation type $S$. Branch and select are sets of labelled session types, whose labels have indices ranging over a non-empty set $I$. Branch $\&\{l_i : S_i\}_{i \in I}$ indicates an external choice, namely what is offered, and it is a generalisation of the input type in which the continuation $S_i$ *depends* on the received label $l_i$. Dually, select $\oplus\{l_i : S_i\}_{i \in I}$ indicates an internal choice, where only one of the available labels $l_i$'s will be chosen, and it is a generalisation of the output type.

Session processes range over $P, Q$. The output process $x!\langle v \rangle.P$ sends a value $v$ on channel endpoint $x$ and continues as $P$; the input process $x?(y).P$ receives on $x$ a value to substitute for the placeholder $y$ in the continuation $P$. The selection process $x \triangleleft l_j.P$ selects label $l_j$ on channel $x$ and proceeds as $P$. The branching process $x \triangleright \{l_i : P_i\}_{i \in I}$ offers a range of labelled alternative processes on channel $x$. The session restriction construct $(\nu xy)P$ creates a session channel, more precisely its two endpoints $x$ and $y$, and binds them in $P$. As usual, the term $\mathbf{0}$ denotes a terminated process and $P \mid Q$ the parallel composition of $P$ and $Q$.

**Duality** Session type duality is a key ingredient in session types theory as it is necessary for communication safety. Two processes willing to communicate, e.g., the client and the server in the equality test, must first agree on a session protocol. Intuitively, client and server should perform dual operations: when one process sends, the other receives, when one offers, the other chooses. Hence, the dual of an input must be an output, the dual of branch must be a select, and vice versa. Formally, duality on session types is defined as the following function:

$$\begin{array}{rcl} \overline{\mathtt{end}} & \triangleq & \mathtt{end} \\ \overline{!T.S} & \triangleq & ?T.\overline{S} \\ \overline{?T.S} & \triangleq & !T.\overline{S} \\ \overline{\oplus\{l_i : S_i\}_{i\in I}} & \triangleq & \&\{l_i : \overline{S}_i\}_{i\in I} \\ \overline{\&\{l_i : S_i\}_{i\in I}} & \triangleq & \oplus\{l_i : \overline{S}_i\}_{i\in I} \end{array}$$

The static checks performed by the typing rules make sure that the peer endpoints of the same session channel have dual types. In particular, this is checked in the restriction rule (T-RES) below:

$$\text{(T-RES)}\quad \frac{\Gamma, x : T, y : \overline{T} \vdash P}{\Gamma \vdash (\nu xy)P}$$

**Standard $\pi$-calculus.** The syntax and reduction semantics for the standard $\pi$-calculus are shown in Figure 2. We use $t$ to range over standard $\pi$-types, to distinguish them from types $T$ and session types $S$, given in the previous paragraph. We also use the notation $\widetilde{\cdot}$ to indicate (finite) sequences of elements. Standard $\pi$-types specify the *capabilities* of channels. The type $\emptyset[]$ is assigned to a channel without any capability, which cannot be used for any input/output action. Standard types $\ell_{\mathtt{i}}[\widetilde{t}]$ and $\ell_{\mathtt{o}}[\widetilde{t}]$ are assigned to channels used *exactly once* to receive and to send a sequence of values of type $\widetilde{t}$, respectively. The variant type $\langle l_i \_ t_i \rangle_{i\in I}$ is a labelled form of disjoint union of types $t_i$ whose indices range over a set $I$.

Linear types and variant types are essential in the encoding of session types. The addition of variant types, as of any structured type, implies the addition of a constructor in the grammar for values, to produce variant values of the form $l\_v$, and of a destructor in the grammar for processes, to consume variant values. Such a destructor is represented by the term **case** $v$ **of** $\{l_i \_ (x_i) \triangleright P_i\}_{i\in I}$, offering different behaviours depending on which variant value $l\_v$ is received and binding $v$ to the corresponding $x_i$. In the operational semantics, the reduction rule in which a variant value is consumed (R$\pi$-CASE) is sometimes called *case normalisation*. Unlike the session $\pi$-calculus, the standard $\pi$-calculus has just one restriction operator that acts on single names, as in $(\nu x)P$.

## 4 Encoding sessions

In this section we present the encoding of session $\pi$-calculus types and terms into linear $\pi$-calculus types and terms, together with the main technical results, following Dardha et al. [14, 15].

**Type encoding.** The encoding of session types into linear $\pi$-types is shown at the top of Figure 3. Types produced by grammar $T$ are encoded in a homomorphic way, e.g., $[\![\sharp T]\!] \triangleq \sharp[\![T]\!]$. The encoding of `end` is a channel with no capabilities $\emptyset[]$ that cannot be used further. Type $?T.S$ is encoded as the linear input channel type carrying a pair of values whose types are the encodings of $T$ and of $S$. The encoding of $!T.S$ is similar except that the type of the second component of the pair is the encoding of $\overline{S}$, since it

$$
\begin{array}{rcll}
[\![\mathtt{end}]\!] & \triangleq & \emptyset[] & \text{(E-End)} \\
[\![!T.S]\!] & \triangleq & \ell_{\mathtt{o}}[[\![T]\!],[\![\overline{S}]\!]] & \text{(E-Out)} \\
[\![?T.S]\!] & \triangleq & \ell_{\mathtt{i}}[[\![T]\!],[\![S]\!]] & \text{(E-Inp)} \\
[\![\oplus\{l_i : S_i\}_{i\in I}]\!] & \triangleq & \ell_{\mathtt{o}}[\langle l_i\_[\![\overline{S_i}]\!]\rangle_{i\in I}] & \text{(E-Select)} \\
[\![\&\{l_i : S_i\}_{i\in I}]\!] & \triangleq & \ell_{\mathtt{i}}[\langle l_i\_[\![S_i]\!]\rangle_{i\in I}] & \text{(E-Branch)}
\end{array}
$$

$$
\begin{array}{rcll}
[\![x]\!]_f & \triangleq & f_x & \text{(E-Name)} \\
[\![\star]\!]_f & \triangleq & \star & \text{(E-Star)}
\end{array}
$$

$$
\begin{array}{rcll}
[\![\mathbf{0}]\!]_f & \triangleq & \mathbf{0} & \text{(E-Inaction)} \\
[\![x!\langle v\rangle.P]\!]_f & \triangleq & (\nu c) f_x!\langle [\![v]\!]_f, c\rangle.[\![P]\!]_{f\{x\mapsto c\}} & \text{(E-Output)} \\
[\![x?(y).P]\!]_f & \triangleq & f_x?(y,c).[\![P]\!]_{f\{x\mapsto c\}} & \text{(E-Input)} \\
[\![x \triangleleft l_j.P]\!]_f & \triangleq & (\nu c) f_x!\langle l_j\_c\rangle.[\![P]\!]_{f\{x\mapsto c\}} & \text{(E-Selection)} \\
[\![x \triangleright \{l_i : P_i\}_{i\in I}]\!]_f & \triangleq & f_x?(y).\ \mathbf{case}\ y\ \mathbf{of}\ \{l_i\_(c) \triangleright [\![P_i]\!]_{f\{x\mapsto c\}}\}_{i\in I} & \text{(E-Branching)} \\
[\![P \mid Q]\!]_f & \triangleq & [\![P]\!]_f \mid [\![Q]\!]_f & \text{(E-Composition)} \\
[\![(\nu xy)P]\!]_f & \triangleq & (\nu c)[\![P]\!]_{f\{x,y\mapsto c\}} & \text{(E-Restriction)} \\
[\![(\nu x)P]\!]_f & \triangleq & (\nu x)[\![P]\!]_f & \text{(E-New)}
\end{array}
$$

Figure 3: Encoding of types, values and processes.

describes the type of a channel as it will be used by the receiver process. The branch and the select types are encoded as linear input and linear output channels carrying variant types having labels $l_i$ and types that are respectively the encoding of $S_i$ and the encoding of $\overline{S_i}$ for all $i \in I$. Again, the reason for using duality of the continuation in the encoding of the select type is the same as for the output type, as select is a generalisation of output type.

**Process encoding.** The encoding of session processes into standard $\pi$-processes is shown at the bottom of Figure 3. The encoding of a process $P$ is parametrised by a function $f$ from channel names to channel names. We say that $f$ is a *renaming function for* $P$ if, for all the names $x$ that occur free in $P$, either $f(x) = x$ or $f(x)$ is a fresh name not occurring in $\mathtt{n}(P)$, where $\mathtt{n}(P)$ is the set of all names of $P$, both free and bound. Also, $f$ is the identity function on all bound names of $P$. Hereafter we write $\mathtt{dom}(f)$ for the domain of $f$ and $f_x$ as an abbreviation for $f(x)$. During the encoding of a session process, its renaming function $f$ is progressively updated. For example, we write $f\{x \mapsto c\}$ or $f\{x,y \mapsto c\}$ for the update of $f$ such that the names $x$ and $y$ are associated to $c$. The notion of a renaming function is extended also to values as expected. In the uses of the definition of the renaming function $f$ for $P$ (respectively $v$), process $P$ (respectively value $v$) will be typed in a typing context, say $\Gamma$. It is implicitly assumed that the fresh names used by $f$ (that is, the names $y$ such that $y = f(x)$, for some $x \neq y$) are also fresh for $\Gamma$.

The motivation for parametrising the encoding of processes and values with a renaming function stems from the key idea of encoding a structured communication over a session channel as a chain of one-shot communications over *linear* channels. Whenever we transmit some payload on a linear channel, the payload is paired with a fresh continuation channel on which the rest of the communication takes place. Such continuation, being fresh, is different from the original channel. Thus, the renaming

function allows us to keep track of this fresh name after each communication.

We now provide some more details on the encoding of terms. Values are encoded as expected, so that a channel name $x$ is encoded to $f_x$ and the $\star$ unit value is encoded to itself. This encoding is trivially extended to every ground value added to the language. In the encoding of the output process, a new channel name $c$ is created and is sent together with the encoding of the payload $v$ along the channel $f_x$. The encoding of the continuation process $P$ is parametrised by an updated $f$ where the name $x$ is associated to $c$. Similarly, the input process listens on channel $f_x$ and receives a pair whose first element (the payload) replaces the name $y$ and whose second element (the continuation channel $c$) replaces $x$ in the continuation process by means of the updated renaming function $f\{x \mapsto c\}$. As indicated in Section 3, session restriction $(\nu xy)P$ creates two fresh names and binds them in $P$ as the opposite endpoints of the same session channel. This is not needed in the standard $\pi$-calculus. The restriction construct $(\nu x)P$ creates and binds a unique name $x$ in $P$; this name identifies both endpoints of the communicating channel. The encoding of a session restriction process $(\nu xy)P$ is a standard channel restriction process $(\nu c)[\![P]\!]_{f\{x,y\mapsto c\}}$ with the new name $c$ used to substitute both $x$ and $y$ in the encoding of $P$. Selection $x \triangleleft l_j.P$ is encoded as the process that first creates a new channel $c$ and then sends on $f_x$ a variant value $l_j\_c$, where $l_j$ is the selected label and $c$ is the channel created to be used for the continuation of the session. The encoding of branching receives on $f_x$ a value, typically being a variant value, which is the guard of the **case** process. According to the transmitted label, one of the corresponding processes $[\![P_i]\!]_{f\{x\mapsto c\}}$ for $i \in I$ will be chosen. Note that the name $c$ is bound in any process $[\![P_i]\!]_{f\{x\mapsto c\}}$. The encoding of the other process constructs, namely inaction, standard channel restriction, and parallel composition, acts as a homomorphism.

**Example 4.1** (Equality test)**.** *We illustrate the encoding of session types and terms on the equality test from the introduction. Thus we also make use of boolean and integer values, and simple operations on them, whose addition to the encoding is straightforward.*

*The encoding of the server's session type S is*

$$[\![S]\!] = \ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{o}}[\mathtt{Bool}, \emptyset[]]]]$$

*while that of the client's session type $\overline{S}$ is*

$$[\![\overline{S}]\!] = \ell_{\mathtt{o}}[\mathtt{Int}, \ell_{\mathtt{i}}[\mathtt{Int}, \ell_{\mathtt{o}}[\mathtt{Bool}, \emptyset[]]]]$$

*Note how the encoding of dual session types boils down to linear channel types that have the* same *payload and dual* outermost *capabilities $\ell_{\mathtt{i}}[\cdot]$ and $\ell_{\mathtt{o}}[\cdot]$. This property holds in general and can be exploited to express the (complex) notion of session type duality in terms of the (simple) property of type equality, as we will see in Section 6.*

*The server process, communicating on endpoint x of type S, is*

$$server \triangleq x?(z_1).x?(z_2).x!\langle z_1 == z_2\rangle.\mathbf{0}$$

*and the client process, communicating on endpoint y of type $\overline{S}$, is*

$$client \triangleq y!\langle 3\rangle.y!\langle 5\rangle.y?(eq).\mathbf{0}$$

*Then we have*

$$\begin{aligned}[\![server]\!]_{\{x\mapsto s\}} &= s?(z_1,c).[\![x?(z_2).x!\langle z_1 == z_2\rangle.\mathbf{0}]\!]_{\{x\mapsto c\}} \\ &= s?(z_1,c).c?(z_2,c').(\nu c'')c'!\langle z_1 == z_2, c''\rangle.\mathbf{0}\end{aligned}$$

*Similarly,*

$$[\![client]\!]_{\{y\mapsto s\}} = (\nu c)s!\langle 3,c\rangle.(\nu c')c!\langle 5,c'\rangle.c'?(eq,c'').\mathbf{0}$$

*The whole server-client system is thus encoded as follows, using $\emptyset$ for the identity function.*

$$[\![(\nu xy)(server \mid client)]\!]_{\emptyset} = (\nu s)[\![(server \mid client)]\!]_{\{x,y\mapsto s\}} = (\nu s)\left([\![server]\!]_{\{x\mapsto s\}} \mid [\![client]\!]_{\{y\mapsto s\}}\right)$$

*(The update $\{x,y \mapsto s\}$ reduces to $\{x \mapsto s\}$ on the server and to $\{y \mapsto s\}$ on the client because they do not contain occurrences of y and x respectively.)*

**Correctness of the encoding.** The presented encoding can be considered as a semantics of session types and session terms. The following theoretical results show that indeed we can derive the typing judgements and the properties of the $\pi$-calculus with sessions via the encoding and the corresponding properties of the linear $\pi$-calculus.

First, the correctness of an encoded typing judgement on the target terms implies the correctness of the judgement on the source terms, and conversely. Similar results hold for values. The encoding is extended to session typing contexts in the expected manner.

**Theorem 4.2** (Type correctness)**.** *The following properties hold:*

1. *If $\Gamma \vdash P$, then $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$ for some renaming function $f$ for $P$;*
2. *If $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$ for some renaming function $f$ for $P$, then $\Gamma \vdash P$.*

Theorem 4.2, and more precisely its proof [15, 12], shows that the encoding can be actually used to reconstruct the typing rules of session types. That is, the typing rules for an operator op of the session $\pi$-calculus can be 'read back' from the typing of the encoding of op.

Next we recall the operational correctness of the encoding. That is, the property that the encoding allows one to faithfully reconstruct the behaviour of a source term from that of the corresponding target term. We recall that $\rightarrow$ is the reduction relation of the two calculi. We write $\hookrightarrow$ for the extension of the structural congruence $\equiv$ with a *case normalisation* indicating the decomposition of a variant value (Section 3).

**Theorem 4.3** (Operational correspondence)**.** *Let $P$ be a session process, $\Gamma$ a session typing context, and $f$ a renaming function for $P$ such that $[\![\Gamma]\!]_f \vdash [\![P]\!]_f$. Then the following statements hold.*

1. *If $P \rightarrow P'$, then $[\![P]\!]_f \rightarrow\!\!\hookrightarrow [\![P']\!]_f$.*
2. *If $[\![P]\!]_f \rightarrow Q$, then there is a session process $P'$ such that*
   - *either $P \rightarrow P'$;*
   - *or there are $x$ and $y$ such that $(\nu xy)P \rightarrow P'$*

   *and $Q \hookrightarrow [\![P']\!]_f$.*

Statement 1 of the above theorem tells us that the reduction of an encoded process mimics faithfully the reduction of the source process, modulo structural congruence or case normalisation. Statement 2 of the theorem tells us that if the encoding of a process $P$ reduces to the encoding of a process $P'$ (via some intermediate process $Q$), then the source process $P$ will reduce directly to $P'$ or it might need a wrap-up under restriction. The reason for the latter is that in the session $\pi$-calculus [53], reduction only occurs under restriction and cannot occur along free names. In particular, in the theorem, $f$ is a generic renaming function; this function could map two free names, say $x$ and $y$, onto the same name; in this case, an input at $x$ and an output at $y$ in the source process could not produce a reduction, whereas they might in the target process.

The two theorems above allow us to derive, as a straightforward corollary, the subject reduction property for the session calculus.

**Corollary 4.4** (Session Subject Reduction). *If* $\Gamma \vdash P$ *and* $P \to Q$*, then* $\Gamma \vdash Q$.

Other properties of the session $\pi$-calculus can be similarly derived from corresponding properties of the standard $\pi$-calculus. For instance, since the encoding respects structural congruence (that is, $P \equiv P'$ if and only if $[\![P]\!]_f \equiv [\![P']\!]_f$), we can derive the invariance of typing under structural congruence in the session $\pi$-calculus.

**Corollary 4.5** (Session Structural Congruence). *If* $\Gamma \vdash P$ *and* $P \equiv P'$*, then also* $\Gamma \vdash P'$.

# 5 Extensions

In this section we discuss several extensions for the presented encoding, which have been proposed in order to accommodate the additional features of subtyping, polymorphism, recursion, higher-order communication and multiparty interactions.

**Subtyping.** Subtyping is a relation between types based on a notion of substitutability. If $T$ is a subtype of $T'$, then any channel of type $T$ can be safely used in a context where a channel of type $T'$ is expected. In the standard $\pi$-calculus, subtyping originates from *capability types* — the possibility of distinguishing the input and output usage of channels [43, 46]. (This is analogous to what happens in languages with references, where capabilities are represented by the read and write usages.) Precisely, the input channel capability is co-variant, whereas the output channel capability is contra-variant in the types of values transmitted (the use of capabilities is actually necessary with linear types, as reported in Figure 2). Subtyping can then be enhanced by means of the variant types, which are co-variant both in depth and in breadth. In the case of session $\pi$-calculus, subtyping must be dealt with also at the level of session types [22]; for instance, branch and select are both co-variant in depth, whereas they are co-variant and contra-variant in breadth, respectively. This duplication of effort can become heavy, particularly when types are enriched with other constructs (a good example are recursive types). The encoding of session types naturally accommodates subtyping, indeed subtyping of the standard $\pi$-calculus can be used to derive subtyping on session types. Writing $<:$ and $\leq$ for, respectively, subtyping for session types and for standard $\pi$-types, for instance we have:

**Theorem 5.1** (Encoding for Subtyping). $T <: T'$ *if and only if* $[\![T]\!] \leq [\![T']\!]$.

**Polymorphism and Higher-Order Communication.** *Polymorphism* is a common and useful type abstraction in programming languages, as it allows operations that are generic by using an expression with several types. Parametric polymorphism has been studied in the standard $\pi$-calculus [46], and in the $\pi$-calculus with session types [4]; for bounded polymorphism in session $\pi$-calculus see Gay [21].

The *Higher-Order* $\pi$*-calculus* (HO$\pi$) models mobility of processes that can be sent and received and thus can be run locally [46]. Higher-order communication for the session $\pi$-calculus [39] has the same benefits as for the $\pi$-calculus, in particular, it models code mobility in a distributed scenario.

Extensions of the encoding to support polymorphism and HO$\pi$ have been studied in [14, 15, 12] and used to test its robustness. The syntax of types and terms is extended to accommodate the new constructs. For polymorphism, session types and standard $\pi$-types are extended with a *type variable* $X$ and with *polymorphic types* $\langle X;T\rangle$ and $\langle X;t\rangle$, respectively. For higher-order communication, session types and standard $\pi$-types are extended with the functional type $T \to \sigma$, assigned to a functional term that can be used without any restriction, and with the linear functional type $T \xrightarrow{1} \sigma$ that must be used exactly once. Correspondingly, the syntax of processes is extended to accommodate the *unpacking* process

(**open** $v$ **as** $(X;x)$ **in** $P$) to deal with polymorphism, and with call-by-value $\lambda$-calculus primitives, namely *abstraction* ($\lambda x:T.P$) and *application* ($PQ$), to deal with higher-order communication.

The encoding of the new type and process constructs is a homomorphism in all cases. Consequently, the proof cases added to Theorems 4.2 and 4.3 are trivial.

**Recursion.** The encoding was also extended to accommodate recursive types and replicated processes by Dardha [10]. Here, the new added types are a recursive type $\mu X.T$ and a type variable $X$, as well as the replicated process $*P$. Recursive (session) types are required to be *guarded*, meaning that in $\mu X.T$, variable $X$ may occur free in $T$ only under at least one of the other type constructs. The paper uses a new duality function, called *complement*, which is inspired by the work of Bernardi et al. [2, 1]. Some new cases for the encoding of recursive session types and processes are:

$$\begin{array}{ll} [\![X]\!] & \triangleq X \\ [\![\mu X.S]\!] & \triangleq \mu X.[\![S]\!] \\ [\![*P]\!]_f & \triangleq *[\![P]\!]_f \end{array}$$

The extended encoding is proved to be sound and complete with respect to typing and reduction (aka operational correspondence). We refer the interested reader to [10, 11].

**Multiparty Session Types.** Multiparty Session Types (MPSTs) [28, 29] accommodate communications between more than two participants. Since their introduction, they have become a major area of investigation within the session type community. Their meta-theory is more complex than that of the binary case, and it is beyond the scope of this paper to revise it in detail.

The core syntax of *multiparty session types* is given by the following grammar

$$\begin{array}{rll} S & ::= & \mathtt{end} \mid X \mid \mu X.S \quad \text{(termination, type variable, recursive type)} \\ & & \mathtt{p}\oplus_{i\in I}!l_i(U_i).S_i \quad \text{(select towards role p)} \\ & & \mathtt{p}\&_{i\in I}?l_i(U_i).S_i \quad \text{(branch from role p)} \end{array}$$

$$B ::= \mathtt{Unit} \mid \ldots \quad \text{(base type)} \qquad U ::= B \mid S \text{ (closed under } \mu) \quad \text{(payload type)}$$

where selection and branching types are annotated with *roles* identifying the participant of a multiparty session to which a message is sent or from which a message is expected. The message consists of a label $l_i$ and a payload of type $U_i$, whereas the continuation $S_i$ indicates how the session endpoint is meant to be used afterwards.

A multiparty session type describes the behaviour of a participant of a multiparty session with respect to all the other participants it interacts with, identified by their role in the session type. In order to obtain the behaviour of a participant with respect to another *particular* participant of the multiparty session, say q, the multiparty session type must be *projected* onto q. Hereafter, we write $S\upharpoonright\mathtt{q}$ for the *partial projection of S onto* q, referring to [47, 48] for its precise definition. Projection yields a type defined by the following syntax, which resembles that of binary session types:

$$\begin{array}{rll} H & ::= & \mathtt{end} \mid X \mid \mu X.H \quad \text{(termination, type variable, recursive type)} \\ & & \oplus_{i\in I}!l_i(U_i).H_i \quad \text{(select)} \\ & & \&_{i\in I}?l_i(U_i).H_i \quad \text{(branch)} \end{array}$$

Projection is a key feature of MPSTs as it is needed in the technical development of a sound type system. At the same time, it also provides a hook by which multiparty sessions and multiparty session

types can be encoded in the standard $\pi$-calculus through the encoding of (binary) session types that we have outlined in Section 3.

Let us briefly comment on the encoding of MPST into linear types given by Scalas et al. [47, 48]. This encoding is fully fledged as it covers the whole MPST and it preserves the theory's *distributivity*. Previous work by Caires and Pérez [3] presents an encoding of MPST into binary session types via a *medium* process, which acts as an orchestrator for the encoding, thus losing distributivity. In the encoding of Scalas et al. no orchestrator is used, hence the encoding preserves its intended choreographic nature as opposed to being orchestrated.

The encoding of a multiparty session type from Scalas et al. is formally defined as:

$$[\![S]\!] \triangleq [\mathtt{p} : [\![S \upharpoonright \mathtt{p}]\!]]_{\mathtt{p} \in S}$$

resulting in a *record* of types with an entry *for each role* $\mathtt{p}$ occurring in the multiparty session type $S$. The encoding of a projected type, namely $[\![S \upharpoonright \mathtt{p}]\!]$, can then be obtained by suitably adapting the function defined in Figure 3. The main cases are summarised below, and the encoding is a homomorphism for the other constructs in the syntactic category $H$ presented above.

$$\begin{aligned} [\![\oplus_{i \in I} !l_i(U_i).H_i]\!] &\triangleq \ell_{\mathtt{o}}[\langle l_i\_([\![U_i]\!], [\![H_i]\!])\rangle_{i \in I}] \\ [\![\&_{i \in I} ?l_i(U_i).H_i]\!] &\triangleq \ell_{\mathtt{i}}[\langle l_i\_([\![U_i]\!], [\![H_i]\!])\rangle_{i \in I}] \end{aligned}$$

The encoding of processes is quite complex and beyond the scope of this paper. The interested reader may refer to Scalas et al. [47, 49] for the formal details and a Scala implementation of multiparty sessions based on this encoding. The encoding of MPST into linear types satisfies several properties, including duality and subtyping preservation, correctness of the encoding with respect to typing, operational correspondence and deadlock freedom preservation. These properties are given in Section 6 of [47].

# 6 Applications

The encoding from session types to linear channel types can be thought of as a way of "explaining" a high-level type language in terms of a simpler, lower-level type language. Protocols written in the lower-level type language tend to be more cumbersome and less readable than the session types they encode. For this reason, it is natural to think of the encoding as nothing more than a theoretical study. Yet, as we are about to see in this section, the very same encoding has also enabled (or at least inspired) further advancements in the theory and practice of session types.

## 6.1 A Type System for Deadlock Freedom

A well-typed session $\pi$–calculus process (and equivalently a well-typed standard $\pi$–calculus one) enjoys communication safety (no message with unexpected type is ever exchanged) but not deadlock freedom. For example, both the session $\pi$–calculus process

$$(\nu x_1 x_2)(\nu y_1 y_2)\,(x_1?(z).y_1!\langle z\rangle.\mathbf{0} \mid y_2?(w).x_2!\langle w\rangle.\mathbf{0}) \tag{1}$$

and the standard $\pi$–calculus process

$$(\nu x)(\nu y)\,(x?().y!\langle\rangle.\mathbf{0} \mid y?().x!\langle\rangle.\mathbf{0}) \tag{2}$$

are well-typed in the respective typing disciplines, but the behaviours they describe on the two sessions/channels are intermingled in such a way that no communication can actually occur: the input from each session/channel must be completed in order to perform the output on the other session/channel.

Several type systems that ensure deadlock freedom in addition to communication safety have been studied for session and standard typed $\pi$-calculi. In a particular line of work, Kobayashi [30, 32] has studied a typing discipline that associates *priorities* to channel types so as to express, at the type level, the relative order in which channels are used, thus enabling the detection of circular dependencies, such as the one shown above. Later on, Padovani [41] has specialised this technique for the linear $\pi$–calculus and, as an effect of the encoding illustrated in Section 3, for the session $\pi$–calculus as well. To illustrate the technique, in this section we consider a refinement of the linear input/output types in Figure 2 as follows

$$t ::= \ell_{\mathtt{o}}[\widetilde{t}]^m \mid \ell_{\mathtt{i}}[\widetilde{t}]^n \mid \cdots$$

where $m$ and $n$ are integers representing priorities: the smaller the number, the higher the priority with which the channel must be used. In the process (2) above, we could assign the types $\ell_{\mathtt{i}}[]^m$ and $\ell_{\mathtt{o}}[]^n$ to respectively $x$ and $y$ on the lhs of $|$ and the types $\ell_{\mathtt{o}}[]^m$ and $\ell_{\mathtt{i}}[]^n$ to respectively $x$ and $y$ on the rhs of $|$. Note that each channel is assigned two types having *dual polarities* (each channel is used in complementary ways on the two sides of $|$) and the *same priority*. Then, the type system imposes constraints on priorities to reflect the order in which channels are used: on the lhs of $|$ we have the constraint $m < n$ since the input on $x$ (with priority $m$) blocks the output on $y$ (with priority $n$); on the rhs of $|$ the opposite happens, resulting in the constraint $n < m$. Obviously, these two constraints are not simultaneously satisfiable, hence the process as a whole is ruled out as ill typed.

In such simple form, this technique fails to deal with most recursive processes. We illustrate the issue through the following server process that computes the factorial of a natural number, in which we use a few standard extensions (replication, conditional, numbers and their operations) to the calculus introduced earlier.

$$*\mathit{fact}?(x,y).\mathbf{if}\ x = 0\ \mathbf{then}\ y!\langle 1\rangle\ \mathbf{else}\ (\nu z)\,(\mathit{fact}!\langle x-1, z\rangle \mid z?(k).y!\langle x \times k\rangle) \tag{3}$$

The server accepts requests on a shared channel *fact*. Each request carries a natural number $x$ and a linear channel $y$ on which the factorial of $x$ is sent as response. The modelling follows the standard recursive definition of the factorial function. In particular, in the recursive case a fresh linear channel $z$ is created from which the factorial $k$ of $x-1$ is received. At that point, the factorial $x \times k$ of $x$ can be sent on $y$. Now assume, for the sake of illustration, that $m$ and $n$ are the priorities associated with $y$ and $z$, respectively. Since $z$ is used in the same position as $y$ in the recursive invocation of *fact*, we expect that $z$ and $y$ should have the same type hence the same priority $m = n$. This clashes with the input on $z$ that blocks the output on $y$, requiring $n < m$. The key observation we can make in order to come up with a more flexible handling of priorities is that a replicated process like (3) above cannot have any *free* linear channel. In fact, the only free channel *fact* is a shared one whereas $y$ is received by the process and $z$ is created within the process. As a consequence, the absolute value of the priorities $m$ and $n$ we associate with $y$ and $z$ does not matter (as long as they satisfy the constraint $n < m$) and they can vary from one request to another. In more technical terms, this corresponds to making *fact polymorphic* in the priority of the channel $y$ received from it and allowing a (priority-limited) form of *polymorphic recursion* when we type outputs such as $\mathit{fact}!\langle x-1, z\rangle$.

It must be pointed out that a process such as (3) is in the scope of Kobayashi's type systems [32]. The additional expressiveness resulting from priority polymorphism enables the successful analysis of recursive processes that interleave actions on different linear channels also in cyclic network topologies. We

do not showcase these more complex scenarios in this brief survey, instead referring the interested reader to [41] for an exhaustive presentation of the technique and to [42] for a proof-of-concept implementation.

As a final remark, it is interesting to note that this technique can be retrofitted to a calculus with native sessions, but it was born in the context of the standard $\pi$–calculus, which features a more primitive communication model. The point is that, in the standard $\pi$–calculus, sequential communications are encoded in a continuation-passing style, meaning that higher-order channels are the norm rather than the exception. So, the quest for expressive type systems ensuring (dead)lock freedom in the standard $\pi$–calculus could not ignore this feature, and this necessity has been a major source of inspiration for the support of priority polymorphism. In this direction, Carbone et al. [8] study (dead)lock freedom for session $\pi$-processes using the encoding from Section 4 and the technique from [32] and show that this combined technique is more fine-grained than other ones adopted in session $\pi$-calculi. Dardha and Pérez [16] present a full account of the deadlock freedom property in session $\pi$-calculi, and compare deadlock freedom obtained by using the encoding and the work from [32] to linear logic approaches, which are used as a yardstick for deadlock freedom.

## 6.2 Session Type Inference

A major concern regarding all type systems is their realisability and applicability in real-world programming languages. In this respect, session type systems pose at least three peculiar challenges: (1) the fact that session endpoints must be treated as *linear resources* that cannot be duplicated or discarded; (2) the need to *update* the session type associated with a session endpoint each time the endpoint is used; (3) the need to express *session type duality* constraints in addition to the usual *type equality* constraints. The first challenge can be easily dealt with only in those (few) languages that provide native support for linear (or at least affine) types. Alternatively, it is possible to devise mechanisms that detect linearity (or affinity) violations at runtime with a modest overhead. The second challenge can be elegantly addressed by adopting a *functional* API for sessions [24], whereby each function/method using a session endpoint returns (possibly along with other results) the same endpoint with its type suitably updated. The last challenge, which is the focus of this section, is a subtle one since session type duality is a complex relation that involves the whole structure of two session types. In fact, it has taken quite some time even just to *correctly define* duality in the presence of recursive session types [2, 23].

Somewhat surprisingly, the encoding of session types into linear channel types allows us to cope with this challenge in the most straightforward way, simply by *getting rid of it*. In Example 4.1 we have shown two session types, one dual of the other, whose respective encodings are *equal* except for the outermost capabilities. This property holds in general.

**Proposition 6.1.** *Let $\overline{\cdot}$ be the partial involution on types such that $\overline{\mathbf{0}[]} = \mathbf{0}[]$ and $\overline{\ell_{\mathtt{i}}[\widetilde{t}]} = \ell_{\mathtt{o}}[\widetilde{t}]$ and $\overline{\ell_{\mathtt{o}}[\widetilde{t}]} = \ell_{\mathtt{i}}[\widetilde{t}]$. Then $[\![\overline{S}]\!] = \overline{[\![S]\!]}$ for every $S$.*

In fact, it is possible to devise a slightly different representation of capabilities so that (session) type duality can be expressed solely in terms of type equality. To this aim, let $\circ$ and $\bullet$ be any two types which we use to represent the absence and presence of a given capability, respectively. We do not need any particular property of $\circ$ and $\bullet$ except the fact that they must be different. In fact, they need not even be inhabited. Now, we can devise a slightly different syntax for linear channel types, as follows:

$$t ::= \ell_{\kappa,\kappa}[\widetilde{t}] \mid \cdots \qquad \kappa ::= \circ \mid \bullet$$

The idea is that a linear channel type carries two separate input and output capabilities (hereafter ranged over by $\kappa$ and $\iota$), each of which can be either present or absent. For example, $\ell_{\circ,\circ}[]$ would be

the same as $\emptyset[]$, $\ell_{\bullet,\circ}[\widetilde{t}]$ would be the same as $\ell_{\mathtt{i}}[\widetilde{t}]$ and $\ell_{\circ,\bullet}[\widetilde{t}]$ would be the same as $\ell_{\mathtt{o}}[\widetilde{t}]$. With this representation of linear channel types the dual of a type can be defined simply as $\overline{\ell_{\kappa,\iota}[\widetilde{t}]} = \ell_{\iota,\kappa}[\widetilde{t}]$, where the input/output capabilities are swapped. Now, suppose that we wish to express a duality constraint $S = \overline{T}$ stating that $S$ is the dual of $T$ and let $\ell_{\kappa,\iota}[\widetilde{s}] = [\![S]\!]$ and $\ell_{\kappa',\iota'}[\widetilde{t}] = [\![T]\!]$ be the encodings of $S$ and $T$, respectively. Using Proposition 6.1 and the revised representation of linear channel types we obtain

$$S = \overline{T} \iff \kappa = \iota' \wedge \iota = \kappa' \wedge \widetilde{s} = \widetilde{t}$$

thereby turning a session type duality constraint into a conjunction of type equality constraints.

This apparently marginal consequence of using encoded (as opposed to native) session types makes it possible to rely on completely standard features of conventional type systems to express and infer complex structural relations on session types. In particular, it allows any Hindley-Milner type inference algorithm to perform *session type inference*. `FuSe` [42] is a library implementation of session types for OCaml that showcases this idea at work. The library supports higher-order sessions, recursive session types and session subtyping by piggybacking on OCaml's type system. Clearly, the inferred (encoded) session types are not as readable as the native ones. This may pose problems in the presence of type errors. To address this issue, the library is accompanied by an external tool called `Rosetta` that decodes encoded session types and pretty prints them as native ones using the inverse of the encoding function $[\![\cdot]\!]$.[1] On similar lines, Scalas and Yoshida [50] develop `lchannels`, a Scala library for session types fully based on the encoding of session types into linear types. As a result, the structure of a session type is checked statically by analysing its encoding onto channel types in Scala, while linearity is checked dynamically at run time as in `FuSe`, as Scala has no support for linearity.

## References


[1] Giovanni Bernardi, Ornela Dardha, Simon J. Gay & Dimitrios Kouzapas (2014): *On Duality Relations for Session Types*. In: *TGC*, *LNCS* 8902, Springer, pp. 51–66, doi:10.1007/978-3-662-45917-1_4.

[2] Giovanni Bernardi & Matthew Hennessy (2014): *Using Higher-Order Contracts to Model Session Types (Extended Abstract)*. In: *CONCUR*, *LNCS* 8704, Springer, pp. 387–401, doi:10.1007/978-3-662-44584-6_27.

[3] Luís Caires & Jorge A. Pérez (2016): *Multiparty Session Types Within a Canonical Binary Theory, and Beyond*. In Elvira Albert & Ivan Lanese, editors: *FORTE*, *LNCS* 9688, Springer, pp. 74–95, doi:10.1007/978-3-319-39570-8_6.

[4] Luís Caires, Jorge A. Pérez, Frank Pfenning & Bernardo Toninho (2013): *Behavioral Polymorphism and Parametricity in Session-Based Communication*. In: *ESOP*, *LNCS* 7792, Springer, pp. 330–349, doi:10.1007/978-3-642-37036-6_19.

[5] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In: *Proc. of CONCUR 2010*, *Lecture Notes in Computer Science* 6269, Springer, pp. 222–236, doi:10.1007/978-3-642-15375-4_16.

[6] Luís Caires, Frank Pfenning & Bernardo Toninho (2014): *Linear Logic Propositions as Session Types*. *MSCS*, doi:10.1017/S0960129514000218.

[7] Sara Capecchi, Mario Coppo, Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou & Elena Giachino (2009): *Amalgamating sessions and methods in object-oriented languages with generics*. *Theor. Comput. Sci.* 410(2-3), pp. 142–167, doi:10.1016/j.tcs.2008.09.016.


[1] The source code of `FuSe` and `Rosetta` is publicly available at `https://github.com/boystrange/FuSe`.

[8] Marco Carbone, Ornela Dardha & Fabrizio Montesi (2014): *Progress as Compositional Lock-Freedom*. In: *COORDINATION*, *LNCS* 8459, Springer, pp. 49–64, doi:10.1007/978-3-662-43376-8_4.

[9] Marco Carbone, Kohei Honda & Nobuko Yoshida (2007): *Structured Communication-Centred Programming for Web Services*. In: *ESOP*, *LNCS* 4421, Springer, pp. 2–17, doi:10.1007/978-3-540-71316-6_2.

[10] Ornela Dardha (2014): *Recursive Session Types Revisited*. In: *BEAT*, *EPTCS* 162, pp. 27–34, doi:10.4204/EPTCS.162.4.

[11] Ornela Dardha (2014): *Recursive Session Types Revisited*. http://www.dcs.gla.ac.uk/~ornela/my_papers/D14-Extended.pdf.

[12] Ornela Dardha (2016): *Type Systems for Distributed Programs: Components and Sessions*. *Atlantis Studies in Computing* 7, Springer / Atlantis Press, doi:10.2991/978-94-6239-204-5.

[13] Ornela Dardha & Simon J. Gay (2018): *A New Linear Logic for Deadlock-Free Session-Typed Processes*. In Christel Baier & Ugo Dal Lago, editors: *FOSSACS*, *LNCS* 10803, Springer, pp. 91–109, doi:10.1007/978-3-319-89366-2_5.

[14] Ornela Dardha, Elena Giachino & Davide Sangiorgi (2012): *Session types revisited*. In: *PPDP*, ACM, New York, NY, USA, pp. 139–150, doi:10.1145/2370776.2370794.

[15] Ornela Dardha, Elena Giachino & Davide Sangiorgi (2017): *Session types revisited*. *Inf. Comput.* 256, pp. 253–286, doi:10.1016/j.ic.2017.06.002.

[16] Ornela Dardha & Jorge A. Pérez (2022): *Comparing type systems for deadlock freedom*. *J. Log. Algebraic Methods Program.* 124, p. 100717, doi:10.1016/j.jlamp.2021.100717.

[17] Romain Demangeon & Kohei Honda (2011): *Full Abstraction in a Subtyped pi-Calculus with Linear Types*. In: *CONCUR*, *LNCS* 6901, Springer, pp. 280–296, doi:10.1007/978-3-642-23217-6_19.

[18] Mariangiola Dezani-Ciancaglini, Elena Giachino, Sophia Drossopoulou & Nobuko Yoshida (2007): *Bounded Session Types for Object Oriented Languages*. In: *FMCO*, *LNCS* 4709, Springer, pp. 207–245, doi:10.1007/978-3-540-74792-5_10.

[19] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida & Sophia Drossopoulou (2006): *Session Types for Object-Oriented Languages*. In: *ECOOP 2006*, *LNCS* 4067, Springer, pp. 328–352, doi:10.1007/11785477_20.

[20] Simon Fowler, Wen Kokke, Ornela Dardha, Sam Lindley & J. Garrett Morris (2021): *Separating Sessions Smoothly*. In Serge Haddad & Daniele Varacca, editors: *CONCUR*, *LIPIcs* 203, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 36:1–36:18, doi:10.4230/LIPIcs.CONCUR.2021.36.

[21] Simon J. Gay (2008): *Bounded polymorphism in session types*. *Mathematical Structures in Computer Science* 18(5), pp. 895–930, doi:10.1017/S0960129508006944.

[22] Simon J. Gay & Malcolm Hole (2005): *Subtyping for session types in the pi calculus*. *Acta Inf.* 42(2-3), pp. 191–225, doi:10.1007/s00236-005-0177-z.

[23] Simon J. Gay, Peter Thiemann & Vasco T. Vasconcelos (2020): *Duality of Session Types: The Final Cut*. In Stephanie Balzer & Luca Padovani, editors: *PLACES@ETAPS*, *EPTCS* 314, pp. 23–33, doi:10.4204/EPTCS.314.3.

[24] Simon J. Gay & Vasco Thudichum Vasconcelos (2010): *Linear type theory for asynchronous session types*. *J. Funct. Program.* 20(1), pp. 19–50, doi:10.1017/S0956796809990268.

[25] Carl Hewitt (1977): *Viewing Control Structures as Patterns of Passing Messages*. *Artif. Intell.* 8(3), pp. 323–364, doi:10.1016/0004-3702(77)90033-9.

[26] Kohei Honda (1993): *Types for Dyadic Interaction*. In: *CONCUR*, *LNCS* 715, Springer, pp. 509–523, doi:10.1007/3-540-57208-2_35.

[27] Kohei Honda, Vasco Vasconcelos & Makoto Kubo (1998): *Language primitives and type disciplines for structured communication-based programming*. In: *ESOP*, *LNCS* 1381, Springer, pp. 22–138, doi:10.1007/BFb0053567.

[28] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In: *POPL*, 43(1), ACM, pp. 273–284, doi:10.1145/1328438.1328472.

[29] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty asynchronous session types*. *Journal of the ACM* 63(1), p. 9, doi:10.1145/2827695.

[30] Naoki Kobayashi (2002): *A Type System for Lock-Free Processes*. *Inf. Comput.* 177(2), pp. 122–159, doi:10.1006/inco.2002.3171.

[31] Naoki Kobayashi (2002): *Type Systems for Concurrent Programs*. In: *10th Anniversary Colloquium of UNU/IIST*, pp. 439–453, doi:10.1007/978-3-540-40007-3_26.

[32] Naoki Kobayashi (2006): *A New Type System for Deadlock-Free Processes*. In: *CONCUR*, *LNCS* 4137, Springer, pp. 233–247, doi:10.1007/11817949_16.

[33] Naoki Kobayashi (2007): *Type Systems for Concurrent Programs*. Available at http://www.kb.ecei.tohoku.ac.jp/~koba/papers/tutorial-type-extended.pdf. Extended version of [31], Tohoku University.

[34] Naoki Kobayashi, Benjamin C. Pierce & David N. Turner (1999): *Linearity and the pi-calculus*. *ACM Trans. Program. Lang. Syst.* 21(5), pp. 914–947, doi:10.1145/330249.330251.

[35] Wen Kokke & Ornela Dardha (2021): *Deadlock-free session types in linear Haskell*. In Jurriaan Hage, editor: *Haskell*, ACM, pp. 1–13, doi:10.1145/3471874.3472979.

[36] Wen Kokke & Ornela Dardha (2021): *Prioritise the Best Variation*. In Kirstin Peters & Tim A. C. Willemse, editors: *FORTE*, *LNCS* 12719, Springer, pp. 100–119, doi:10.1007/978-3-030-78089-0_6.

[37] Sam Lindley & J. Garrett Morris (2016): *Embedding session types in Haskell*. In: *Proc. of Haskell*, ACM, pp. 133–145, doi:10.1145/2976002.2976018.

[38] Fabrizio Montesi & Nobuko Yoshida (2013): *Compositional Choreographies*. In: *CONCUR*, *LNCS* 8052, Springer, pp. 425–439, doi:10.1007/978-3-642-40184-8_30.

[39] Dimitris Mostrous & Nobuko Yoshida (2007): *Two Session Typing Systems for Higher-Order Mobile Processes*. In: *TLCA*, *LNCS* 4583, Springer, pp. 321–335, doi:10.1007/978-3-540-73228-0_23.

[40] Dominic Orchard & Nobuko Yoshida (2017): *Session Types with Linearity in Haskell*. *Behavioural Types: from Theory to Tools*, pp. 219–242, doi:10.13052/rp-9788793519817.

[41] Luca Padovani (2014): *Deadlock and lock freedom in the linear π-calculus*. In Thomas A. Henzinger & Dale Miller, editors: *CSL-LICS*, ACM, pp. 72:1–72:10, doi:10.1145/2603088.2603116.

[42] Luca Padovani (2017): *Type-Based Analysis of Linear Communications*. In Simon Gay & António Ravara, editors: *Behavioural Types: from Theory to Tools*, River Publishers, pp. 193–217, doi:10.13052/rp-9788793519817.

[43] Benjamin C. Pierce & Davide Sangiorgi (1993): *Typing and Subtyping for Mobile Processes*. In: *LICS*, IEEE Computer Society, pp. 376–385, doi:10.1109/LICS.1993.287570.

[44] Riccardo Pucella & Jesse A. Tov (2008): *Haskell session types with (almost) no class*. In: *Proc. of Haskell*, ACM, doi:10.1145/1411286.1411290.

[45] Davide Sangiorgi (1998): *An Interpretation of Typed Objects into Typed pi-Calculus*. *Inf. Comput.* 143(1), pp. 34–73, doi:10.1006/inco.1998.2711.

[46] Davide Sangiorgi & David Walker (2001): *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press.

[47] Alceste Scalas, Ornela Dardha, Raymond Hu & Nobuko Yoshida (2017): *A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming*. In Peter Müller, editor: *ECOOP*, *LIPIcs* 74, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 24:1–24:31, doi:10.4230/LIPIcs.ECOOP.2017.24.

[48] Alceste Scalas, Ornela Dardha, Raymond Hu & Nobuko Yoshida (2017): *A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming*. Technical Report 2, Imperial College London. Available at https://www.doc.ic.ac.uk/research/technicalreports/2017/#2.

[49] Alceste Scalas, Ornela Dardha, Raymond Hu & Nobuko Yoshida (2017): *A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming (Artifact)*. Dagstuhl Artifacts Ser. 3(2), pp. 03:1–03:2, doi:10.4230/DARTS.3.2.3.

[50] Alceste Scalas & Nobuko Yoshida (2016): *Lightweight Session Programming in Scala*. In Shriram Krishnamurthi & Benjamin S. Lerner, editors: *ECOOP*, *LIPIcs* 56, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 21:1–21:28, doi:10.4230/LIPIcs.ECOOP.2016.21.

[51] Kaku Takeuchi, Kohei Honda & Makoto Kubo (1994): *An Interaction-based Language and its Typing System*. In: *PARLE*, *LNCS* 817, Springer, pp. 398–413, doi:10.1007/3-540-58184-7_118.

[52] Antonio Vallecillo, Vasco Thudichum Vasconcelos & António Ravara (2006): *Typing the Behavior of Software Components using Session Types*. Fundam. Inform. 73(4), pp. 583–598. Available at https://content.iospress.com/articles/fundamenta-informaticae/fi73-4-07.

[53] Vasco T. Vasconcelos (2012): *Fundamentals of session types*. Information Computation 217, pp. 52–70, doi:10.1016/j.ic.2012.05.002.

[54] Vasco Thudichum Vasconcelos, Simon J. Gay & António Ravara (2006): *Type checking a multithreaded functional language with session types*. Theor. Comput. Sci. 368(1-2), pp. 64–87, doi:10.1016/j.tcs.2006.06.028.

[55] Philip Wadler (2012): *Propositions as sessions*. In: *ICFP*, ACM, pp. 273–286, doi:10.1145/2364527.2364568.

[56] Nobuko Yoshida & Vasco Thudichum Vasconcelos (2007): *Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication*. Electr. Notes Theor. Comput. Sci. 171(4), pp. 73–93, doi:10.1016/j.entcs.2007.02.056.

# The Way We Were: Structural Operational Semantics Research in Perspective

Luca Aceto
Department of Computer Science,
Reykjavik University,
Reykjavik, Iceland
Gran Sasso Science Institute,
L'Aquila, Italy
luca@ru.is luca.aceto@gssi.it

Pierluigi Crescenzi
Gran Sasso Science Institute,
L'Aquila, Italy
pierluigi.crescenzi@gssi.it

Anna Ingólfsdóttir
Department of Computer Science,
Reykjavik University,
Reykjavik, Iceland
annai@ru.is

Mohammad Reza Mousavi
Department of Informatics,
King's College London
London, UK
mohammad.mousavi@kcl.ac.uk

## 1 Introduction

This position paper on the (meta-)theory of Structural Operational Semantic (SOS) is motivated by the following two questions:

- Is the (meta-)theory of SOS dying out as a research field?
- If so, is it possible to rejuvenate this field with a redefined purpose?

In this article, we will consider possible answers to those questions by first analysing the history of the EXPRESS/SOS workshops and the data concerning the authors and the presentations featured in the editions of those workshops as well as their subject matters.

The first International Workshop on Structural Operation was held in London, UK in 2004. The workshop was established as 'a forum for researchers, students and practitioners interested in new developments, and directions for future investigation, in the field of structural operational semantics. One of the specific goals of the workshop was to establish synergies between the concurrency and programming language communities working on the theory and practice of SOS.' At its ninth edition, the SOS workshop joined forces with the nineteenth edition of International Workshop on Expressiveness in Concurrency. The joint workshop was meant to cover the broader scope of 'the formal semantics of systems and programming concepts, and on the expressiveness of mathematical models of computation.'

We examined the contributions dedicated to the theory of SOS presented in the EXPRESSS/SOS workshop series (and, prior to that, in the SOS workshop) and whether they appeared before or after the merger between the EXPRESS and SOS workshops. We also used the collected data to compute a well-established measure of similarity between the two phases in the life of the SOS workshop, before and after the merger with EXPRESS. Beyond these data- and graph-mining analyses, we reflect on the major results developed in nearly four decades of research on SOS and identify, in our admittedly biased opinion, its strengths and gaps.

G. Caltais and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2023).
EPTCS 387, 2023, pp. 26–40, doi:10.4204/EPTCS.387.3

The results of our quantitative and qualitative analyses all indicate a diminishing interest in the theory of SOS as a field of research. Even though 'all good things must come to an end', we strive to finish this position paper on an upbeat note by addressing our second motivating question with some optimism. To this end, we use our personal reflections and an analysis of recent trends in two of the flagship conferences in the field of Programming Languages (namely POPL and PDLI) to draw some conclusions on possible future directions that may rejuvenate research on the (meta-)theory of SOS. We hope that our musings will entice members of the research community to breathe new life into a field of research that has been kind to three of the authors of this article.

**Whence this collaboration?** This article is the result of a collaboration between a researcher from the theory of algorithms and their applications, Pierluigi Crescenzi, and three contributors to the theory of SOS. Pierluigi Crescenzi has recently offered data- and graph-mining analyses of conferences such as CONCUR, in cooperation with Luca Aceto in [4], SIROCCO [24] and ICALP—see the presentation available at `https://slides.com/piluc/icalp-50?token=fl3BBJ8j`. All authors thought that it was natural to combine quantitative data- and graph-mining analysis techniques with qualitative domain-specific knowledge to offer a fairly well-rounded perspective on the developments in the (meta-)theory of SOS and its relation to the SOS and EXPRESS/SOS workshops. Both the Java code and the Julia software developed by Pierluigi Crescenzi, which was used for the quantitative analyses reported in this article and the aforementioned earlier ones, are publicly available at the following GitHub repository: `https://github.com/piluc/ConferenceMining`. We encourage everyone interested in carrying out data- and graph-mining analyses of conferences to use it!

## 2 Data Collection and Analysis

To set the stage for our reflections on the (meta-)theory of SOS, we have carried out some data analysis on the SOS and EXPRESS/SOS workshops.

### 2.1 Data Collection

We extracted the following data from all the eleven past editions of the joint EXPRESS/SOS workshop:

1. the authors and titles of contributed talks;
2. invited speakers and the titles of their presentations or papers;
3. the number of submissions and accepted papers; and
4. at least two and at most three subject matter classifiers from the scope of EXPRESS/SOS.

Much of the gathered data was extracted from the tables of contents and proceedings of those editions of the workshop, which are all available in open access form as volumes of Electronic Proceedings in Computer Science (EPTCS), and from the DBLP page devoted to the Workshop on Structural Operational Semantics. In case of missing information regarding the number of submissions, we approached the workshops chairs and gathered that information through personal communication. For subject matter classification, since the general classifications, such as the one by the ACM, were too general for our purposes, we manually read the abstract (and in a few cases full papers) and identified domain-specific classifiers, using the scope definition of the EXPRESS/SOS workshop.

The results of our data collection are publicly available online.

The choice of focusing our analysis on the last eleven editions was motivated by the fact that, since 2012, the SOS workshop decided to join forces with the EXPRESS workshop and created a new joint venue. This gave us a consistent view of how the topics featured in the joint workshop have evolved over time and of how (structural) operational semantics has been represented in the joint workshop since 2012. However, using the data we collected, we also took the opportunity to compare the two phases of the SOS workshop, the first as an independent workshop in the period 2004–2011 and the second as EXPRESS/SOS from 2012 till 2022.

## 2.2 Automatic Analysis

Based on the articles that were archived in the workshop proceedings, we found that

- 194 authors contributed articles to the workshop proceedings since 2004;
- 90 colleagues published papers in the proceedings of the first eight editions of the SOS workshop;
- 122 researchers contributed articles to the joint EXPRESS/SOS workshop in the period 2012–2022;
- 18 authors published papers in the SOS workshop proceedings both before and after the merger with the EXPRESS workshop, which means that there were 104 contributors to EXPRESS/SOS who had never published in the SOS workshop in the period 2004–2011.

The above-mentioned data allow us to compute a measure of similarity between the two phases of the SOS workshop, before and after the merger with EXPRESS, using the Sørensen-Dice index, which is a statistic used to measure the similarity of two samples. Given two sets $A$ and $B$, the *Jaccard index* $J(A,B)$ is equal to $\frac{|A\cap B|}{|A\cup B|}$, and the *Sørensen-Dice index* is equal to $\frac{2J(A,B)}{1+J(A,B)}$, see [28, 66].

The Sørensen-Dice index for the lists of authors in the two phases of the SOS workshop is roughly 0.17. This value indicates that the SOS workshop is not as similar to the joint EXPRESS/SOS workshop as one might have expected. By way of comparison, quoting from the data- and graph-mining analysis of CONCUR presented in [4],

> the conference that is most similar to CONCUR is LICS (with Sørensen-Dice index approximately equal to 0.3), followed by TACAS (approximately 0.25), CAV (approximately 0.24), and CSL (approximately 0.21).

Computing the Sørensen-Dice index for SOS 2004–2022 and CONCUR, LICS, PLDI and POPL yields low values of similarity, namely 0.106396 (CONCUR), 0.0622966 (LICS), 0.00585138 (PLDI) and 0.0303169 (POPL). This is due to the fact that the sets of authors of those conferences is much larger than that of the SOS workshop, namely 1475 (CONCUR), 1953 (LICS), 3220 (PLDI) and 1979 (POPL).

When quantifying the degree of similarity between a small workshop like SOS with larger conferences, it might be more appropriate to consider the Szymkiewicz–Simpson coefficient (also known as the overlap coefficient) [65, 68, 69, 73]. Given two sets $A$ and $B$, the *Szymkiewicz–Simpson coefficient* is equal to $\frac{|A\cap B|}{\min(|A|,|B|)}$. The values of that coefficient for the conferences we considered above are roughly 0.45 (CONCUR), 0.34 (LICS), 0.05 (PLDI) and 0.17 (POPL). Those values seem to support the view that SOS is rather similar to CONCUR and LICS, has some similarity with POPL, but is very dissimilar to PLDI.

## 2.3 Centrality Measures

The *static graph* (or collaboration graph) of SOS is an undirected graph whose nodes are the authors who presented at least one paper at SOS, and whose edges link two authors who coauthored at least one paper

(not necessarily presented at SOS). In other words, this graph is the subgraph of the DBLP collaboration graph induced by the set of SOS authors.

Centrality measures have been used as a key tool for understanding social networks, such as the static graph of SOS, and are used to assess the 'importance' of a given node in a network—see, for instance, [35]. Therefore, to quantify the role played by authors who have contributed to the SOS workshop, we have computed the following classic centrality measures on the largest connected component of the static graph of SOS.

- Degree: This is the number of neighbours of a node in the graph (that is, the number of coauthors).
- Closeness: This is the average distance from one author to all other authors of its connected component.
- Betweenness: This is the fraction of shortest paths, passing through one author, between any pair of other authors in its connected component.

The top ten SOS authors with respect to the above-mentioned three centrality measures are, in decreasing order:

- Degree: Luca Aceto, Anna Ingólfsdóttir, Mohammad Reza Mousavi, Nobuko Yoshida, Rob van Glabbeek, Bas Luttik, Wan Fokkink, Michel Reniers, Catuscia Palamidessi, and Rocco De Nicola.
- Closeness: Luca Aceto, Rob van Glabbeek, Nobuko Yoshida, Matthew Hennessy, Catuscia Palamidessi, Anna Ingólfsdóttir, Rocco De Nicola, Daniele Gorla, Bas Luttik, and Uwe Nestmann.
- Betweenness: Luca Aceto, Matthew Hennessy, Nobuko Yoshida, Rob van Glabbeek, Rocco De Nicola, Catuscia Palamidessi, Daniele Gorla, Frank de Boer, Bartek Klin, and Uwe Nestmann.

In addition, we also calculated the *temporal closeness*, which is an analogue of closeness that takes the number of years of a collaboration between two authors into account—see the paper [25] for more information on this centrality measure. The top ten SOS authors according to temporal closeness are, in decreasing order: Luca Aceto, Anna Ingólfsdóttir, Wan Fokkink, Rocco De Nicola, Catuscia Palamidessi, Bas Luttik, Michel Reniers, Rob van Glabbeek, Jan Friso Groote, and Mohammad Reza Mousavi.

Finally, to get a glimpse of the evolution of the aforementioned measures of similarity and centrality in the two phases of the SOS workshop, we computed them on the static graphs before and after the merger with EXPRESS.

Before the merger with EXPRESS, the 2004–2011 editions of SOS had Szymkiewicz–Simpson index approximately of 0.42 with CONCUR, 0.37 with LICS, 0.067 with PLDI and 0.2 with POPL. After the merger with EXPRESS, those figures become 0.512 for CONCUR, 0.352 for LICS, 0.032 for PLDI and 0.152 for POPL. So, from 2012 onwards, SOS has become more similar to CONCUR and even more dissimilar to PLDI and POPL than before.

The top ten authors at the SOS workshop also change before and after the merger. When focusing on the period before the merger, the most central authors are as follows, in decreasing order:

- Degree: Luca Aceto, Michel Reniers, Mohammad Reza Mousavi, Anna Ingólfsdóttir, Wan Fokkink, Rocco De Nicola, José Meseguer, Rob van Glabbeek, Catuscia Palamidessi, and David de Frutos-Escrig.
- Closeness: Luca Aceto, Anna Ingólfsdóttir, Rocco De Nicola, Rob van Glabbeek, Matthew Hennessy, Georgiana Caltais, Mohammad Reza Mousavi, Eugen-Ioan Goriac, Michel Reniers, and Catuscia Palamidessi.

- Betweenness: Rocco De Nicola, Luca Aceto, Catuscia Palamidessi, José Meseguer, Frank de Boer, Filippo Bonchi, Matthew Hennessy, Michel Reniers, Rob van Glabbeek, and David de Frutos-Escrig.
- Temporal closeness: Luca Aceto, Anna Ingólfsdóttir, Wan Fokkink, Michel Reniers, Mohammad Reza Mousavi, José Meseguer, Jan Friso Groote, Rob van Glabbeek, Rocco De Nicola, and Catuscia Palamidessi.

After the merger with EXPRESS, our graph-mining analysis yields the following most central authors, in decreasing order:

- Degree: Nobuko Yoshida, Luca Aceto, Bas Luttik, Rob van Glabbeek, Mohammad Reza Mousavi, Uwe Nestmann, Anna Ingólfsdóttir, Jorge Pérez, Jos Baeten, and Hans Hüttel.
- Closeness: Nobuko Yoshida, Luca Aceto, Rob van Glabbeek, Catuscia Palamidessi, Anna Ingólfsdóttir, Bas Luttik, Uwe Nestmann, Mohammad Reza Mousavi, Iain Phillips, and Mariangiola Dezani-Ciancaglini.
- Betweenness: Nobuko Yoshida, Rob van Glabbeek, Daniele Gorla, Luca Aceto, Bas Luttik, Bartek Klin, Uwe Nestmann, Catuscia Palamidessi, Hans Hüttel, and Rance Cleaveland.
- Temporal closeness: Luca Aceto, Anna Ingólfsdóttir, Bas Luttik, Tim Willemse, Catuscia Palamidessi, Mohammad Reza Mousavi, Jos Baeten, Jan Friso Groote, Jorge Pérez, and Rob van Glabbeek.

## 2.4 The Two Lives of the SOS Workshop

As we saw above, the first and the second life of the SOS workshop are not that similar after all, which seems to indicate that the eleven joint editions of the EXPRESS/SOS workshop were more about expressiveness than about structural operational semantics[1]. To see whether this is really the case, we visually summarise the data we collected in Figure 1 and provide its details below:

- The proceedings of EXPRESS/SOS 2012 included 10 papers, five of which dealt with topics related to operational semantics and its mathematical (meta-)theory—that's 50% of the articles and the largest percentage of SOS contributions to EXPRESS/SOS in the period 2012–2022.
- The proceedings of EXPRESS/SOS 2013 included seven papers, two of which dealt with topics related to operational semantics and its mathematical (meta-)theory—that's 28.5% of the contributions .
- The proceedings of EXPRESS/SOS 2014 included eight papers, two of which (25%) dealt with topics related to the theory of structural operational semantics.
- The proceedings of EXPRESS/SOS 2015 included six papers, one of which (16.7%) dealt with topics related to the theory of structural operational semantics.
- The proceedings of EXPRESS/SOS 2016 included five papers, none of which dealt mainly with operational semantics.

[1] Another possible explanation for the low degree of similarity between the pre- and post-merger incarnations of the SOS workshop is that the community welcomed many new authors from 2012 onwards. This would be a healthy and welcome development and is, in fact, supported by the data we collected. However, the analysis we present in what follows gives some indication that, since 2014, the scientific programme of EXPRESS/SOS has featured only a few papers on structural operational semantics.

- The proceedings of EXPRESS/SOS 2017 included six papers, one of which (16.7%) dealt mainly with operational semantics.
- The proceedings of EXPRESS/SOS 2018 included seven papers, none of which dealt mainly with operational semantics.
- The proceedings of EXPRESS/SOS 2019 included seven papers, two of which 28.5% dealt mainly with operational semantics.
- The proceedings of EXPRESS/SOS 2020 included six papers, none of which dealt mainly with operational semantics.
- The proceedings of EXPRESS/SOS 2021 included six papers, none of which dealt mainly with operational semantics.
- The proceedings of EXPRESS/SOS 2022 included eight papers, none of which dealt mainly with operational semantics.

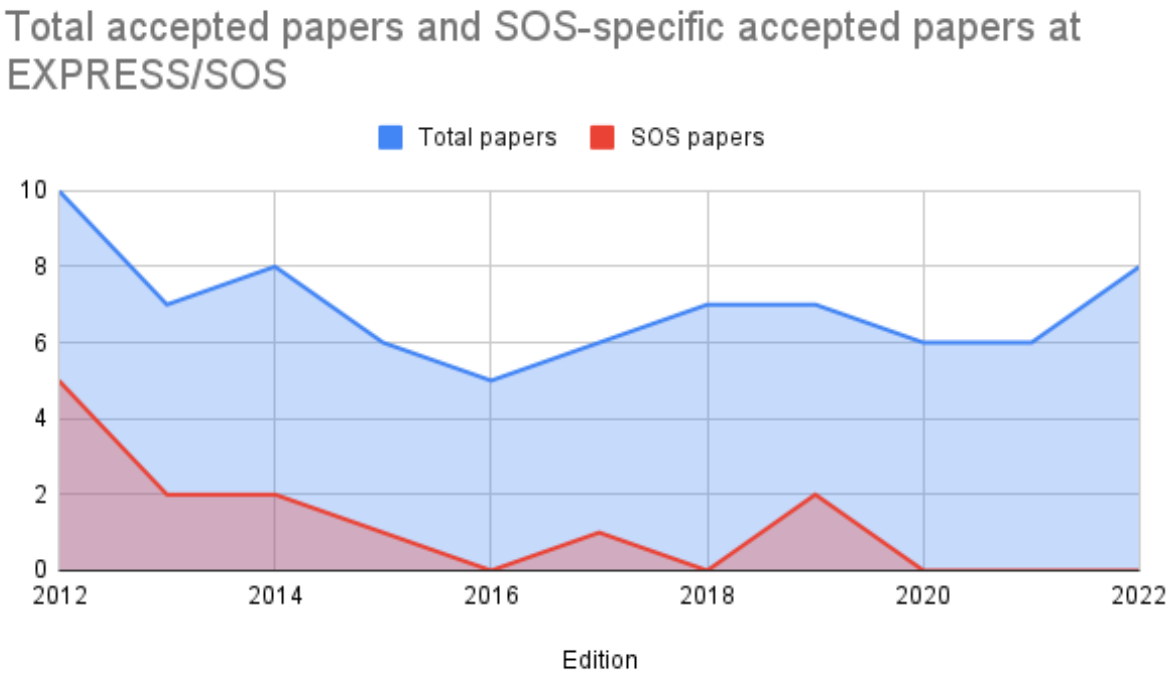



Figure 1: Total number of accepted paper (blue) and the number of accepted papers on SOS theory at the EXPRESS/SOS Workshop since 2012.

So, only 13 out of the 76 papers published in the proceedings of EXPRESS/SOS since 2012 dealt with topics in SOS theory (17.1% of published papers). In passing, we also note that 16 out of the 110 presentations at the workshop in the period 2012–2022 were devoted to topics in SOS theory (that is, 14.5% of the workshop presentations). Research in SOS was well represented at EXPRESS/SOS in the first three editions of the joint workshop. However, five of the last seven instalments of the workshop did not include any presentations devoted to topics that were mainly related to structural operational

semantics. In particular, EXPRESS/SOS 2020–2022 did not have any talks on the theory and applications of structural operational semantics.

### 2.5 Reflections on the Analysis Results

Reading through the EXPRESS/SOS contributions relevant to the theory of SOS reveals that the most recent results mostly focused on two aspects of SOS specifications: foundational aspects concerning the bialgebraic interpretation of SOS due to Turi and Plotkin [70], as well as compositionality of quantitative notions of equivalence such as probabilistic bisimilarity. Below, we provide a more nuanced analysis of this trend.

Another observation is that the diminishing strength in the provision of results on the theory of SOS can be largely attributed to a lack of projects (particularly, PhD studentships) in this area. Almost all of the results on the meta-theory of SOS contributed to the EXPRESS/SOS series had a co-author with a PhD project on this topic. A reduction in the number of doctoral students does not bode well for the healthy development of any research field.

## 3 Personal Reflections

Since the appearance of Plotkin's seminal Aarhus technical report [59], reprinted in slightly revised form as a journal paper in [61] with some historical remarks by Plotkin himself in [60], structural operational semantics has arguably become the most widely used approach to defining the semantics of programming and executable specification languages. To our mind, it is as impactful and popular today as it has been for over forty years. Indeed, one would be hard pressed to find papers on the theory of programming and specification languages that do not use structural operational semantics in some way. Moreover, the semantics of full-blown programming or domain-specific languages is still given in that style, reflecting its flexibility and applicability—see, for instance, the paper [44] for a small-step semantics of full Ethereum-virtual-machine bytecode that is formalised in the $F*$ proof assistant [67] and then validated against the official Ethereum test suite.

As Plotkin highlights in his aforementioned piece on the origins of structural operational semantics, the essence of that approach to semantics is that it is *rule based* and that the rules should be *syntax directed* in order to support compositional language specifications and reasoning, as in the denotational approach to semantics. Conceptually, this rule-based view of operational semantics naturally led to the development of a theory of SOS language specifications that focused on the rules used in semantic definitions. The gist of that line of research, which can be traced back to de Simone's work [64], was to study *rule formats* for operational specifications guaranteeing that every program in the specified language afford some semantic property of interest. So, rule formats offered a way to reduce the checking of semantic properties of programs in a language to syntactic checks on the rules used to define the operational semantics of the language. The literature on what came to be called the 'meta-theory of structural operational semantics' is by now very large and we cannot do it justice in this paper. We refer the interested reader to the survey articles [6, 58] and to the references therein as well as the proceedings of SOS, EXPRESS/SOS, and of conferences such as CONCUR, LICS and POPL, for much more information and recent references. Naturally, since its first edition in 2004, the SOS workshop has served as a venue for the publication of several articles on SOS meta-theory.

Three of the authors of this piece have been amongst the contributors to the development of the fascinating research on rule formats for operational specifications and thoroughly enjoyed doing so.

However, we feel that the time has come for a critical appraisal of the strengths, weaknesses and possible future of that line of research and to speculate about whether the data we discussed in Section 2 reflects the musings we present in the rest of this note.

### 3.1 Strengths

In our, admittedly biased, opinion, research on rule formats for structural operational semantics has led to a wealth of interesting and elegant theoretical results, ranging from those on the meaning of rule-based specifications using rules with negative premises (see, for instance, the articles [13, 40, 18]) to congruence formats for several behavioural equivalences obtained uniformly from their modal characterisations via modal decomposition (see, for example, [11, 34, 32, 33] and the references therein). Early studies of congruence rule formats, such as those reported in the seminal [12, 45], were accompanied by characterisations of the largest congruences included in trace semantics induced by the collection of operators that can be specified in the rule formats studied in those references. After all these years, we still find it amazing that such results could be proved at all!

Below we provide a non-exhaustive list of the available meta-theorems with sufficient strength (more than a single paper, with more than one application to a language) and we refer to the past review papers/chapters [6, 58] for a more exhaustive list to the date of their publication:

- Congruence: proving congruence (compositionality) for various notions of strong [52, 72], weak [32], higher-order [54], data-rich [56], timed [47], and quantitative behavioural equivalences [26, 16, 17]; supporting various syntactic language features such as formal variables and binders [52, 20], as well as semantic features such as negative premises and predicates, terms as labels, and ordering on rules.
- (De-)Compositional reasoning methods: decomposing logical formulae (in the multi-modal $\mu$-calculus, also known as Hennessy-Milner logic with recursion, [49, 50]) according to the semantics of various operators for various notions of bisimilarity [33, 32, 34] and their quantitative extensions [16, 17]; interestingly, this can lead not only to a reasoning method for checking modal formulae, but can also serve as a recipe for 'generating' congruence formats for different notions of equivalence, once their modal characterisation is defined.
- Axiomatisation and algebraic properties: to generate sound and ground-complete axiomatisations for strong bisimilarity [2], as well as weak behavioural equivalences [41], and equivalences with data [37]. An orthogonal line of enquiry considered identifying sufficient conditions guaranteeing various algebraic properties of language operators such as commutativity [57], associativity [23], zero and unit elements [3], and idempotence [1]; we refer to an accessible overview paper [8] summarising such results to its date of publication.

There have been a number of implementations of such results in tools [7, 55, 71], mostly based on rewriting logic [21].

Several of the theorems from the theory of structural operational semantics have found application in the study of process calculi, reducing the need to prove congruence and axiomatisation results, amongst others, from scratch for each calculus and have been extended to settings including, for instance, probabilistic and stochastic features (see, for example, [17, 26]), as well as to higher-order calculi, as in the recent [43]. The article [43] belongs to a fruitful and still active line of research, stemming from the seminal work by Turi and Plotkin [70], providing bialgebraic foundations to the theory of structural operational semantics.

The contributions to the work on rule formats and on the meta-theory of structural operational semantics have striven to achieve a reasonably good trade-off between the generality of the technical results and the ease with which they can be applied to specific languages. Ideally, one would always like to have simple syntactic restrictions on rule formats that guarantee desired semantic properties in a wide variety of applications. Indeed, following a Pareto Principle, very often simple rule formats cover many of the languages of interest and one quickly hits a threshold where complex and hard-to-check definitions are needed to extend the applicability of obtained results. In many cases, the 'curse of generality' led to definitions of rule formats whose constraints are arguably not purely syntactic any more and may even be undecidable. As an example, Klin and Nachyla [48] have shown that it is undecidable whether an operational specification that uses rules with negative premises has a least supported model and whether it has a unique supported model or a stable model. It is also undecidable whether such a specification is complete. As mentioned by Klin and Nachyla in the aforementioned reference, these negative results entail that formats such as the complete ntyft/ntyxt [31] 'are not *bona fide* syntactic formats, as there is no algorithmic way to tell whether a given specification fits such a format.' So, the pursuit of generality is, to our mind, a double-edged sword and can be seen as both a strength and a weakness of several result on rule formats and the meta-theory of structural operational semantics.

In the context of EXPRESS/SOS, we observed that this tradition of strong theoretical results is dying down: from 2012 to 2017, we counted nine contribution to the foundation of SOS specifications [7, 14, 27, 37, 38, 39, 48, 51, 62], including on the bialgebraic framework [14, 48, 62], as well as congruence for quantitative notions of equivalence [27, 38, 39, 51] and axiomatisation results [37]; however, this number dropped to only one contribution from 2018 to 2022 on the meaning of SOS specification and compositionality of equivalences on open terms [42].

In summary, we believe that the study of rule formats and of the meta-theory of structural operational semantics has yielded many elegant results that have been of some use for the working concurrency theorist. However, first, the number of such contributions has significantly dropped in the past few years and, second, one has to wonder whether that line of work has had impact on the field of programming language semantics. We will offer some musings on that question in the coming section.

## 3.2 Gaps

To our mind, apart from its intrinsic scientific interest, the theory of structural operational semantics based on rule formats has served the concurrency-theory community well by providing elegant, and often general and deep, results that have both explained the underlying reasons why specific languages enjoyed several semantic properties and served as tools to prove new theorems as instances of a general framework. The use of 'syntactic' rule formats to establish properties of interest about formal systems has also been used in logic. By way of example, Ciabattoni and Leitsch have given algorithmic conditions guaranteeing that some logics enjoy cut elimination [19]. However, despite its undoubted successes, to our mind, the theory of rule formats has not yet had the impact one might have expected on the community working on the theory of programming languages. Perusing the proceedings of the premier conferences in that field indicates that much of the research on programming-language semantics and its applications is done in the context of proof assistants such as Coq [9, 22][2] and on frameworks built on top of those—see, for instance, the highly influential Iris framework for higher-order concurrent separation logic [46].

We speculate that this relative lack of impact might be due to the fact that the theory of structural

[2]Coq is available at `https://coq.inria.fr/`.

operational semantics based on rule formats has been mostly developed within the process algebra community. This has naturally led to the development of results and frameworks having process calculi as main application area. As a consequence, despite some foundational research [5, 30, 56], the development of a widely-applicable theory of rule formats for languages having first-class notions of data and memory, as well as binding constructs is still somewhat in its infancy. This limits the applicability of the results obtained by the concurrency theory community to mainstream programming languages. Moreover, the software tools embodying the theory of structural operational semantics developed so far have mostly taken the form of prototypes and are arguably not as mature and usable as those produced by groups working on the theory of programming languages [63]. The initial work carried out within the PLanCompS [10] aimed to address this gap based on the Modular SOS framework that has been pioneered by Mosses [53]; this line of work has been influential and has led to other frameworks such as the iCoLa framework for incremental language development [36].

### 3.3 Trends and Opportunities

To relate the past strengths to future trends, particularly regarding emerging application areas of operational semantics, we analysed the table of contents of five past editions of flagship conferences in programming languages: POPL (from 2021 to 2023, inclusive) and PLDI (from 2021 to 2022, inclusive). The aim of the analysis was to find areas where the available strength in the theory of SOS can be exploited. We aimed to be as inclusive as possible and tried to mention any such areas, even if the exploitation of available strength would require a major rework or transformation of ideas and results. Below we provide a raw list of keywords that we encountered in our analysis:

- POPL 2023: Semantics of Probabilistic and Quantum programs, Coq Proof Libraries, Nominal Sets, Co-Algebra and Bisimulation, Multi-Language Semantics, Session types.
- POPL 2022: Session types, Semantics of Probabilistic and Quantum programs, Semantic Substitution and congruence.
- POPL 2021: Semantics of Probabilistic Programs, Nominal Semantics, Hyper-properties and non-interference, functorial semantics
- PLDI 2022: Information flow analysis, equational and algebraic reasoning (also applied to quantum programs), sound sequentialisation, Kleene algebra, language interoperability, verified compilation (also applied to quantum programs).
- PLDI 2021: Language translation conformance and compiler verification, session types, regular expressions, semantics of probabilistic and quantum programs.

In all the POPL and PLDI editions we reviewed, abstract interpretation (also for quantum programs), analysing weak memory models, and reasoning using separation logics are featured prominently.

It appears from our analysis that the following activities may have substantial potential impact:

- semantic meta-theorems about quantitative transition systems (particularly probabilistic and quantum transition systems [15, 29]);
- providing mechanised semantic frameworks, particularly in proof assistants such as Coq;
- defining general semantic frameworks and theorems for different memory models and models of parallelism;
- defining general compositional frameworks for reasoning with separation logics and logics of incorrectness;

- devising algorithms for test-case generation, for instance, for compiler testing, based on a semantic framework.

We hope to see work on some of those topics in the near future, which might lead to a new lease of life for the (meta-)theory of SOS and its applications.

**Acknowledgements** We thank Valentina Castiglioni and Peter Mosses for their comments on a draft of this piece. Luca Aceto and Anna Ingólfsdóttir were partly supported by the projects 'Open Problems in the Equational Logic of Processes (OPEL)' (grant no. 196050) and 'Mode(l)s of Verification and Monitorability (MoVeMent)' (grant no. 217987) of the Icelandic Research Fund. Mohammad Reza Mousavi have been partially supported by the UKRI Trustworthy Autonomous Systems Node in Verifiability, Grant Award Reference EP/V026801/2 and the EPSRC grant on Verified Simulation for Large Quantum Systems (VSL-Q), Grant Award Reference EP/Y005244/1.

## References

[1] Luca Aceto, Arnar Birgisson, Anna Ingólfsdóttir, Mohammad Reza Mousavi & Michel A. Reniers (2012): *Rule formats for determinism and idempotence*. *Science of Computer Programming* 77(7-8), pp. 889–907, doi:10.1016/j.scico.2010.04.002.

[2] Luca Aceto, Bard Bloom & Frits W. Vaandrager (1994): *Turning SOS Rules into Equations*. *Information and Computation* 111(1), pp. 1–52, doi:10.1006/inco.1994.1040.

[3] Luca Aceto, Matteo Cimini, Anna Ingólfsdóttir, Mohammad Reza Mousavi & Michel A. Reniers (2011): *SOS rule formats for zero and unit elements*. *Theoretical Computer Science* 412(28), pp. 3045–3071, doi:10.1016/j.tcs.2011.01.024.

[4] Luca Aceto & Pierluigi Crescenzi (2022): *CONCUR Through Time*. *Bulletin of the EATCS* 138, pp. 157–166. Available at `http://bulletin.eatcs.org/index.php/beatcs/article/view/737`.

[5] Luca Aceto, Ignacio Fábregas, Álvaro García-Pérez, Anna Ingólfsdóttir & Yolanda Ortega-Mallén (2019): *Rule Formats for Nominal Process Calculi*. *Logical Methods in Computer Science* 15(4), pp. 2:1–2:46, doi:10.23638/LMCS-15(4:2)2019.

[6] Luca Aceto, Wan Fokkink & Chris Verhoef (2001): *Structural Operational Semantics*. In Jan A. Bergstra, Alban Ponse & Scott A. Smolka, editors: *Handbook of Process Algebra*, North-Holland / Elsevier, pp. 197–292, doi:10.1016/b978-044482830-9/50021-7.

[7] Luca Aceto, Eugen-Ioan Goriac & Anna Ingólfsdóttir (2013): *Meta SOS - A Maude Based SOS Meta-Theory Framework*. In Johannes Borgström & Bas Luttik, editors: *Proceedings Combined 20th International Workshop on Expressiveness in Concurrency and 10th Workshop on Structural Operational Semantics, EXPRESS/SOS 2013, Buenos Aires, Argentina, 26th August, 2013*, *EPTCS* 120, pp. 93–107, doi:10.4204/EPTCS.120.8.

[8] Luca Aceto, Anna Ingólfsdóttir, Mohammad Reza Mousavi & Michel A. Reniers (2009): *Algebraic Properties for Free!* *Bulletin of the European Association for Theoretical Computer Science (BEATCS)* 99, pp. 81–104.

[9] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series, Springer, doi:10.1007/978-3-662-07964-5.

[10] L. Thomas van Binsbergen, Neil Sculthorpe & Peter D. Mosses (2016): *Tool support for component-based semantics*. In Lidia Fuentes, Don S. Batory & Krzysztof Czarnecki, editors: *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*, ACM, pp. 8–11, doi:10.1145/2892664.2893464.

[11] Bard Bloom, Wan Fokkink & Rob van Glabbeek (2004): *Precongruence formats for decorated trace semantics*. *ACM Transactions on Computational Logic* 5(1), pp. 26–78, doi:10.1145/963927.963929.

[12] Bard Bloom, Sorin Istrail & Albert R. Meyer (1995): *Bisimulation Can't be Traced*. *Journal of the ACM* 42(1), pp. 232–268, doi:10.1145/200836.200876.

[13] Roland N. Bol & Jan Friso Groote (1996): *The Meaning of Negative Premises in Transition System Specifications*. *Journal of the ACM* 43(5), pp. 863–914, doi:10.1145/234752.234756.

[14] Marcello M. Bonsangue, Stefan Milius & Jurriaan Rot (2012): *On the specification of operations on the rational behaviour of systems*. In Bas Luttik & Michel A. Reniers, editors: *Proceedings Combined 19th International Workshop on Expressiveness in Concurrency and 9th Workshop on Structured Operational Semantics, EXPRESS/SOS 2012, Newcastle upon Tyne, UK, September 3, 2012*, *EPTCS* 89, pp. 3–18, doi:10.4204/EPTCS.89.2.

[15] Richard Bornat, Jaap Boender, Florian Kammueller, Guillaume Poly & Rajagopal Nagarajan (2020): *Describing and Simulating Concurrent Quantum Systems*. In Armin Biere & David Parker, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*, *Lecture Notes in Computer Science* 12079, Springer, pp. 271–277, doi:10.1007/978-3-030-45237-7_16.

[16] Valentina Castiglioni, Daniel Gebler & Simone Tini (2018): *SOS-based Modal Decomposition on Non-deterministic Probabilistic Processes*. *Logical Methods in Computer Science* 14(2), doi:10.23638/LMCS-14(2:18)2018.

[17] Valentina Castiglioni & Simone Tini (2020): *Probabilistic divide & congruence: Branching bisimilarity*. *Theoretical Computer Science* 802, pp. 147–196, doi:10.1016/j.tcs.2019.09.037.

[18] Martin Churchill, Peter D. Mosses & Mohammad Reza Mousavi (2013): *Modular Semantics for Transition System Specifications with Negative Premises*. In Pedro R. D'Argenio & Hernán C. Melgratti, editors: *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, *Lecture Notes in Computer Science* 8052, Springer, pp. 46–60, doi:10.1007/978-3-642-40184-8_5.

[19] Agata Ciabattoni & Alexander Leitsch (2008): *Towards an algorithmic construction of cut-elimination procedures*. *Mathematical Structures in Computer Science* 18(1), pp. 81–105, doi:10.1017/S0960129507006573.

[20] Matteo Cimini, Mohammad Reza Mousavi, Michel A. Reniers & Murdoch James Gabbay (2012): *Nominal SOS*. In Ulrich Berger & Michael W. Mislove, editors: *Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2012, Bath, UK, June 6-9, 2012*, *Electronic Notes in Theoretical Computer Science* 286, Elsevier, pp. 103–116, doi:10.1016/j.entcs.2012.08.008.

[21] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer & Carolyn L. Talcott, editors (2007): *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. *Lecture Notes in Computer Science* 4350, Springer, doi:10.1007/978-3-540-71999-1.

[22] Thierry Coquand & Gérard P. Huet (1988): *The Calculus of Constructions*. *Information and Computation* 76(2/3), pp. 95–120, doi:10.1016/0890-5401(88)90005-3.

[23] Sjoerd Cranen, Mohammad Reza Mousavi & Michel A. Reniers (2008): *A Rule Format for Associativity*. In Franck van Breugel & Marsha Chechik, editors: *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, *Lecture Notes in Computer Science* 5201, Springer, pp. 447–461, doi:10.1007/978-3-540-85361-9_35.

[24] Pierluigi Crescenzi (2023): *Thirty Years of SIROCCO A Data and Graph Mining Comparative Analysis of Its Temporal Evolution*. In Sergio Rajsbaum, Alkida Balliu, Joshua J. Daymude & Dennis Olivetti, editors: *Structural Information and Communication Complexity - 30th International Colloquium, SIROCCO 2023, Alcalá de Henares, Spain, June 6-9, 2023, Proceedings*, *Lecture Notes in Computer Science* 13892, Springer, pp. 18–32, doi:10.1007/978-3-031-32733-9_2.

[25] Pierluigi Crescenzi, Clémence Magnien & Andrea Marino (2020): *Finding Top-k Nodes for Temporal Closeness in Large Temporal Graphs*. *Algorithms* 13(9), p. 211, doi:10.3390/a13090211.

[26] Pedro R. D'Argenio, Daniel Gebler & Matias David Lee (2016): *A general SOS theory for the specification of probabilistic transition systems*. *Information and Computation* 249, pp. 76–109, doi:10.1016/j.ic.2016.03.009.

[27] Pedro R. D'Argenio, Matias David Lee & Daniel Gebler (2015): *SOS rule formats for convex and abstract probabilistic bisimulations*. In Silvia Crafa & Daniel Gebler, editors: *Proceedings of the Combined 22th International Workshop on Expressiveness in Concurrency and 12th Workshop on Structural Operational Semantics, EXPRESS/SOS 2015, Madrid, Spain, 31st August 2015*, *EPTCS* 190, pp. 31–45, doi:10.4204/EPTCS.190.3.

[28] Lee Raymond Dice (1945): *Measures of the Amount of Ecologic Association Between Species*. *Ecology* 26(3), pp. 297–302, doi:10.2307/1932409.

[29] Yuan Feng, Yuxin Deng & Mingsheng Ying (2014): *Symbolic Bisimulation for Quantum Processes*. *ACM Transactions on Computational Logic* 15(2), pp. 14:1–14:32, doi:10.1145/2579818.

[30] Marcelo P. Fiore & Sam Staton (2009): *A congruence rule format for name-passing process calculi*. *Information and Computation* 207(2), pp. 209–236, doi:10.1016/j.ic.2007.12.005.

[31] Wan Fokkink & Rob van Glabbeek (1996): *Ntyft/Ntyxt Rules Reduce to Ntree Rules*. *Information and Computation* 126(1), pp. 1–10, doi:10.1006/inco.1996.0030.

[32] Wan Fokkink & Rob van Glabbeek (2017): *Divide and congruence II: From decomposition of modal formulas to preservation of delay and weak bisimilarity*. *Information and Computation* 257, pp. 79–113, doi:10.1016/j.ic.2017.10.003.

[33] Wan Fokkink, Rob van Glabbeek & Bas Luttik (2019): *Divide and congruence III: From decomposition of modal formulas to preservation of stability and divergence*. *Information and Computation* 268, doi:10.1016/j.ic.2019.104435.

[34] Wan Fokkink, Rob van Glabbeek & Paulien de Wind (2012): *Divide and congruence: From decomposition of modal formulas to preservation of branching and $\eta$-bisimilarity*. *Information and Computation* 214, pp. 59–85, doi:10.1016/j.ic.2011.10.011.

[35] Linton C. Freeman (1978–1979): *Centrality in Social Networks Conceptual Clarification*. *Social Networks* 1(3), pp. 215–239, doi:10.1016/0378-8733(78)90021-7.

[36] Damian Frölich & L. Thomas van Binsbergen (2022): *iCoLa: A Compositional Meta-language with Support for Incremental Language Development*. In Bernd Fischer, Lola Burgueño & Walter Cazzola, editors: *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2022, Auckland, New Zealand, December 6-7, 2022*, ACM, pp. 202–215, doi:10.1145/3567512.3567529.

[37] Daniel Gebler, Eugen-Ioan Goriac & Mohammad Reza Mousavi (2013): *Algebraic Meta-Theory of Processes with Data*. In Johannes Borgström & Bas Luttik, editors: *Proceedings Combined 20th International Workshop on Expressiveness in Concurrency and 10th Workshop on Structural Operational Semantics, EXPRESS/SOS 2013, Buenos Aires, Argentina, 26th August, 2013*, *EPTCS* 120, pp. 63–77, doi:10.4204/EPTCS.120.6.

[38] Daniel Gebler & Simone Tini (2013): *Compositionality of Approximate Bisimulation for Probabilistic Systems*. In Johannes Borgström & Bas Luttik, editors: *Proceedings Combined 20th International Workshop on Expressiveness in Concurrency and 10th Workshop on Structural Operational Semantics, EXPRESS/SOS 2013, Buenos Aires, Argentina, 26th August, 2013*, *EPTCS* 120, pp. 32–46, doi:10.4204/EPTCS.120.4.

[39] Daniel Gebler & Simone Tini (2014): *Fixed-point Characterization of Compositionality Properties of Probabilistic Processes Combinators*. In Johannes Borgström & Silvia Crafa, editors: *Proceedings Combined 21st International Workshop on Expressiveness in Concurrency, EXPRESS 2014, and 11th Workshop on Structural Operational Semantics, SOS 2014, Rome, Italy, 1st September 2014*, *EPTCS* 160, pp. 63–78, doi:10.4204/EPTCS.160.7.

[40] Rob van Glabbeek (2004): *The meaning of negative premises in transition system specifications II*. Journal of Logical and Algebraic Methods in Programming 60–61, pp. 229–258, doi:10.1016/j.jlap.2004.03.007.

[41] Rob van Glabbeek (2011): *On cool congruence formats for weak bisimulations*. Theoretical Computer Science 412(28), pp. 3283–3302, doi:10.1016/j.tcs.2011.02.036.

[42] Rob van Glabbeek (2019): *On the Meaning of Transition System Specifications*. In Jorge A. Pérez & Jurriaan Rot, editors: Proceedings Combined 26th International Workshop on Expressiveness in Concurrency and 16th Workshop on Structural Operational Semantics, EXPRESS/SOS 2019, Amsterdam, The Netherlands, 26th August 2019, EPTCS 300, pp. 69–85, doi:10.4204/EPTCS.300.5.

[43] Sergey Goncharov, Stefan Milius, Lutz Schröder, Stelios Tsampas & Henning Urbat (2023): *Towards a Higher-Order Mathematical Operational Semantics*. Proceedings of the ACM on Programming Languages 7(POPL), pp. 632–658, doi:10.1145/3571215.

[44] Ilya Grishchenko, Matteo Maffei & Clara Schneidewind (2018): *A Semantic Framework for the Security Analysis of Ethereum Smart Contracts*. In Lujo Bauer & Ralf Küsters, editors: Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Lecture Notes in Computer Science 10804, Springer, pp. 243–269, doi:10.1007/978-3-319-89722-6_10.

[45] Jan Friso Groote & Frits W. Vaandrager (1992): *Structured Operational Semantics and Bisimulation as a Congruence*. Information and Computation 100(2), pp. 202–260, doi:10.1016/0890-5401(92)90013-6.

[46] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal & Derek Dreyer (2018): *Iris from the ground up: A modular foundation for higher-order concurrent separation logic*. Journal of Functional Programming 28, p. e20, doi:10.1017/S0956796818000151.

[47] Marco Kick (2002): *Coalgebraic Modelling of Timed Processes*. Ph.D. thesis, School of Informatics, University of Edinburgh. Available at `https://www.lfcs.inf.ed.ac.uk/reports/04/ECS-LFCS-04-435/`.

[48] Bartek Klin & Beata Nachyla (2017): *Some undecidable properties of SOS specifications*. Journal of Logical and Algebraic Methods in Programming 87, pp. 94–109, doi:10.1016/j.jlamp.2016.08.005.

[49] Dexter Kozen (1983): *Results on the Propositional mu-Calculus*. Theoretical Computer Science 27, pp. 333–354, doi:10.1016/0304-3975(82)90125-6.

[50] Kim Guldstrand Larsen (1990): *Proof Systems for Satisfiability in Hennessy-Milner Logic with Recursion*. Theoretical Computer Science 72(2&3), pp. 265–288, doi:10.1016/0304-3975(90)90038-J.

[51] Matias David Lee, Daniel Gebler & Pedro R. D'Argenio (2012): *Tree rules in probabilistic transition system specifications with negative and quantitative premises*. In Bas Luttik & Michel A. Reniers, editors: Proceedings Combined 19th International Workshop on Expressiveness in Concurrency and 9th Workshop on Structured Operational Semantics, EXPRESS/SOS 2012, Newcastle upon Tyne, UK, September 3, 2012, EPTCS 89, pp. 115–130, doi:10.4204/EPTCS.89.9.

[52] Cornelis A. Middelburg (2001): *Variable binding operators in transition system specifications*. Journal of Logical and Algebraic Methods in Programming 47(1), pp. 15–45, doi:10.1016/S1567-8326(00)00003-5.

[53] Peter D. Mosses (2004): *Modular structural operational semantics*. Journal of Logical and Algebraic Methods in Programming 60–61, pp. 195–228, doi:10.1016/j.jlap.2004.03.008.

[54] Mohammad Reza Mousavi, Murdoch Gabbay & Michel A. Reniers (2005): *SOS for Higher Order Processes*. In Martín Abadi & Luca de Alfaro, editors: CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings, Lecture Notes in Computer Science 3653, Springer, pp. 308–322, doi:10.1007/11539452_25.

[55] Mohammad Reza Mousavi & Michel A. Reniers (2005): *Prototyping SOS Meta-theory in Maude*. In Peter D. Mosses & Irek Ulidowski, editors: Proceedings of the Second Workshop on Structural Operational Semantics, SOS@ICALP 2005, Lisbon, Portugal, July 10, 2005, Electronic Notes in Theoretical Computer Science 156, Elsevier, pp. 135–150, doi:10.1016/j.entcs.2005.09.030.

[56] Mohammad Reza Mousavi, Michel A. Reniers & Jan Friso Groote (2005): *Notions of bisimulation and congruence formats for SOS with data*. Information and Computation 200(1), pp. 107–147, doi:10.1016/j.ic.2005.03.002.

[57] Mohammad Reza Mousavi, Michel A. Reniers & Jan Friso Groote (2005): *A syntactic commutativity format for SOS*. Information Processing Letters 93(5), pp. 217–223, doi:10.1016/j.ipl.2004.11.007.

[58] Mohammad Reza Mousavi, Michel A. Reniers & Jan Friso Groote (2007): *SOS formats and meta-theory: 20 years after*. Theoretical Computer Science 373(3), pp. 238–272, doi:10.1016/j.tcs.2006.12.019.

[59] Gordon D. Plotkin (1981): *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University.

[60] Gordon D. Plotkin (2004): *The origins of structural operational semantics*. Journal of Logical and Algebraic Methods in Programming 60–61, pp. 3–15, doi:10.1016/j.jlap.2004.03.009.

[61] Gordon D. Plotkin (2004): *A structural approach to operational semantics*. Journal of Logical and Algebraic Methods in Programming 60–61, pp. 17–139, doi:10.1016/j.jlap.2004.05.001.

[62] Jurriaan Rot (2017): *Distributive Laws for Monotone Specifications*. In Kirstin Peters & Simone Tini, editors: Proceedings Combined 24th International Workshop on Expressiveness in Concurrency and 14th Workshop on Structural Operational Semantics, EXPRESS/SOS 2017, Berlin, Germany, 4th September 2017, EPTCS 255, pp. 83–97, doi:10.4204/EPTCS.255.6.

[63] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar & Rok Strnisa (2010): *Ott: Effective tool support for the working semanticist*. J. Funct. Program. 20(1), pp. 71–122, doi:10.1017/S0956796809990293.

[64] Robert de Simone (1985): *Higher-Level Synchronising Devices in Meije-SCCS*. Theoretical Computer Science 37, pp. 245–267, doi:10.1016/0304-3975(85)90093-3.

[65] George Gaylord Simpson (1960): *Notes on the Measurement of Faunal Resemblance*. American Journal of Science, Bradley Volume 258-A, pp. 300–311.

[66] Thorvald Julius Sørensen (1948): *A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on Danish commons*. Kongelige Danske Videnskabernes Selskab 5(4), pp. 1–34. Available at `https://www.royalacademy.dk/Publications/High/295_S%C3%B8rensen,%20Thorvald.pdf`.

[67] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué & Santiago Zanella-Béguelin (2016): *Dependent Types and Multi-Monadic Effects in F**. In: 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM, pp. 256–270, doi:10.1145/2837614.2837655.

[68] Dezydery Szymkiewicz (1926): *Etude Comparative de la Distribution Florale*. Rev. Forest 1.

[69] Dezydery Szymkiewicz (1934): *Une Contribution Statistique a la Géographie Floristique*. Acta Societatis Botanicorum Poloniae 34(3), pp. 249–265, doi:10.5586/asbp.1934.012.

[70] Daniele Turi & Gordon D. Plotkin (1997): *Towards a Mathematical Operational Semantics*. In: Proceedings, 12th Annual IEEE Symposium on Logic in Computer Science, Warsaw, Poland, June 29 - July 2, 1997, IEEE Computer Society, pp. 280–291, doi:10.1109/LICS.1997.614955.

[71] Alberto Verdejo & Narciso Martí-Oliet (2006): *Executable structural operational semantics in Maude*. Journal of Logical and Algebraic Methods in Programming 67(1-2), pp. 226–293, doi:10.1016/j.jlap.2005.09.008.

[72] Chris Verhoef (1995): *A Congruence Theorem for Structured Operational Semantics with Predicates and Negative Premises*. Nordic Journal on Computing 2(2), pp. 274–302.

[73] M. K. Vijaymeena & K. Kavitha (2016): *A Survey on Similarity Measures in Text Mining*. Machine Learning and Applications 3(1), pp. 19–28, doi:10.5121/mlaij.2016.310.

# Comparing Deadlock-Free Session Processes, Revisited (Short Paper)

Channa Dias Perera
University of Groningen (NL)

Jorge A. Pérez
University of Groningen (NL)

We are interested in type systems that enforce the deadlock-freedom property for $\pi$-calculus processes. Several different type systems have been proposed, which consider different dialects of the $\pi$-calculus and use different insights to rule out the circular dependencies that induce deadlocks.

Prior work by Dardha and Pérez rigorously compared two kinds of type systems: (i) type systems based on priorities, as pioneered by Kobayashi; and (ii) type systems based on Curry-Howard interpretations of linear logic propositions as session types. They show that the former subsume the latter, i.e., type systems based on linear logic induce a class of deadlock-free processes that is strictly included in the class induced by priority-based systems. Dardha and Pérez's comparison considers languages with similar (reduction) semantics, which admit the same definition of deadlock-freedom.

In this short paper, we report on ongoing work aimed at extending Dardha and Pérez's classification to consider a class of deadlock-free processes with a *self-synchronizing* transition semantics, which is induced by presentations of linear logic based on *hypersequents*. Integrating this class into a formal comparison is interesting but subtle, as trivially deadlocked-processes in Dardha and Pérez's setting actually enjoy the deadlock-freedom property under the self-synchronizing regime.

G. Caltais and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2023).
EPTCS 387, 2023, pp. 41–41, doi:10.4204/EPTCS.387.4

# A Cancellation Law for Probabilistic Processes

Rob van Glabbeek*
University of Edinburgh
University of New South Wales
rvg@stanford.edu

Jan Friso Groote
Eindhoven University of Technology
j.f.groote@tue.nl

Erik de Vink
Eindhoven University of Technology
evink@win.tue.nl

We show a cancellation property for probabilistic choice. If $\mu \oplus \rho$ and $\nu \oplus \rho$ are branching probabilistic bisimilar, then $\mu$ and $\nu$ are also branching probabilistic bisimilar. We do this in the setting of a basic process language involving non-deterministic and probabilistic choice and define branching probabilistic bisimilarity on distributions. Despite the fact that the cancellation property is very elegant and concise, we failed to provide a short and natural combinatorial proof. Instead we provide a proof using metric topology. Our major lemma is that every distribution can be unfolded into an equivalent stable distribution, where the topological arguments are required to deal with uncountable branching.

## 1 Introduction

A familiar property of the real numbers $\mathbb{R}$ is the additive cancellation law: if $x+z=y+z$ then $x=y$. Switching to the Boolean setting, and interpreting $+$ by $\vee$ and $=$ by $\Leftrightarrow$, the property becomes $(x \vee z) \Leftrightarrow (y \vee z)$ implies $x \Leftrightarrow y$. This is not generally valid. Namely, if $z$ is true, nothing can be derived regarding the truth values of $x$ and $y$. Algebraically speaking, the reals provide an 'additive inverse', and the Booleans do not have a 'disjunctive' version of it.

A similar situation holds for strong bisimilarity in the pure non-deterministic setting vs. strong bisimilarity in the mixed non-deterministic and probabilistic setting. When we have $E+G \leftrightarroweq F+G$ for the non-deterministic processes $E+G$ and $F+G$, it may or may not be the case that $E \leftrightarroweq F$. However, if $P \,{}_{1/2}\!\oplus R \leftrightarroweq Q \,{}_{1/2}\!\oplus R$ for the probabilistic processes $P \,{}_{1/2}\!\oplus R$ and $Q \,{}_{1/2}\!\oplus R$, with probabilistic choice ${}_{1/2}\!\oplus$, we can exploit a semantic characterization of bisimilarity as starting point of a calculation. The characterization reads

$$P \leftrightarroweq Q \quad \text{iff} \quad \forall C \in \mathscr{E}/\!\leftrightarroweq\colon \mu[C] = \nu[C] \tag{1}$$

where the distributions $\mu, \nu \in \mathit{Distr}(\mathscr{E})$ are induced by $P$ and $Q$. To spell out the above, two probabilistic processes $P$ and $Q$ are strongly bisimilar iff the distributions $\mu$ and $\nu$ induced by $P$ and $Q$, respectively, assign the same probability to every equivalence class $C$ of non-deterministic processes modulo strong bisimilarity. In the situation that $P \,{}_{1/2}\!\oplus R \leftrightarroweq Q \,{}_{1/2}\!\oplus R$ we obtain from (1), for equivalence classes $C \in \mathscr{E}/\!\leftrightarroweq$ and distributions $\mu$, $\nu$, and $\rho$ induced by the processes $P$, $Q$, and $R$, that

$$\begin{aligned} P \,{}_{1/2}\!\oplus R \leftrightarroweq Q \,{}_{1/2}\!\oplus R \implies \forall C \in \mathscr{E}/\!\leftrightarroweq\colon \tfrac{1}{2}\mu[C] + \tfrac{1}{2}\rho[C] = \tfrac{1}{2}\nu[C] + \tfrac{1}{2}\rho[C] \implies \\ \forall C \in \mathscr{E}/\!\leftrightarroweq\colon \tfrac{1}{2}\mu[C] = \tfrac{1}{2}\nu[C] \implies \forall C \in \mathscr{E}/\!\leftrightarroweq\colon \mu[C] = \nu[C] \implies P \leftrightarroweq Q \end{aligned}$$

relying on the arithmetic of the reals.

We are interested in whether the cancellation law also holds for weaker notions of process equivalence for probabilistic processes, especially for branching probabilistic bisimilarity as proposed in [16].

*Supported by Royal Society Wolfson Fellowship RSWF\R1\221008

G. Caltais and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2023).
EPTCS 387, 2023, pp. 42–58, doi:10.4204/EPTCS.387.5

We find that it does but the proof is involved. A number of initial attempts were directed towards finding a straightforward combinatorial proof, but all failed. A proof in a topological setting, employing the notion of sequential compactness to deal with potentially infinite sequences of transitions is reported in this paper. We leave the existence of a shorter, combinatorial proof as an open question.

Our strategy to prove the above cancellation law for probabilistic processes and branching probabilistic bisimilarity is based on two intermediate results: (i) every probabilistic process unfolds into a so-called *stable* probabilistic process, and (ii) for stable probabilistic processes a characterization of the form (1) does hold. Intuitively, a stable process is a process that cannot do an internal move without leaving its equivalence class.

In order to make the above more concrete, let us consider an example. For the ease of presentation we use distributions directly, rather than probabilistic processes. Let the distributions $\mu$ and $\nu$ be given by

$$\mu = \tfrac{1}{2}\delta(a\cdot\partial(\mathbf{0}))\oplus\tfrac{1}{2}\delta(b\cdot\partial(\mathbf{0}))$$
$$\nu = \tfrac{1}{3}\delta(\tau\cdot(\partial(a\cdot\partial(\mathbf{0}))\,{}_{\frac{1}{2}}\!\oplus\,\partial(b\cdot\partial(\mathbf{0}))))\oplus\tfrac{1}{3}\delta(a\cdot\partial(\mathbf{0}))\oplus\tfrac{1}{3}\delta(b\cdot\partial(\mathbf{0}))$$

with $a$ and $b$ two different actions. The distribution $\mu$ assigns probability 0.5 to $a\cdot\partial(\mathbf{0})$, meaning an $a$-action followed by a deadlock with probability 1, and probability 0.5 to $b\cdot\partial(\mathbf{0})$, i.e. a $b$-action followed by deadlock with probability 1. The distribution $\nu$ assigns both these non-deterministic processes probability $\frac{1}{3}$ and assigns the remaining probability $\frac{1}{3}$ to $\tau\cdot(\partial(a\cdot\partial(\mathbf{0}))\,{}_{\frac{1}{2}}\!\oplus\,\partial(b\cdot\partial(\mathbf{0})))$, where a $\tau$-action precedes a 50-50 percent choice between the processes mentioned earlier. Below, we show that $\mu$ and $\nu$ are branching probabilistic bisimilar, i.e. $\mu\leftrightarrows_b\nu$. However, if $C_1$, $C_2$ and $C_3$ are the three different equivalence classes of $\tau\cdot(\partial(a\cdot\partial(\mathbf{0}))\,{}_{\frac{1}{2}}\!\oplus\,\partial(b\cdot\partial(\mathbf{0}))$, $a\cdot\partial(\mathbf{0})$ and $b\cdot\partial(\mathbf{0})$, respectively, we have

$$\mu[C_1]=0\neq\tfrac{1}{3}=\nu[C_1],\ \mu[C_2]=\tfrac{1}{2}\neq\tfrac{1}{3}=\nu[C_2],\text{ and }\mu[C_3]=\tfrac{1}{2}\neq\tfrac{1}{3}=\nu[C_3].$$

Thus, although $\mu\leftrightarrows_b\nu$, it does not hold that $\mu[C]=\nu[C]$ for every equivalence class $C$. Note that the distribution $\nu$ is not stable, in the sense that it allows an internal transition to the branchingly equivalent $\nu$.

As indicated, we establish in this paper a cancellation law for branching probabilistic bisimilarity in the context of mixed non-deterministic and probabilistic choice, exploiting the process language of [6], while dealing with distributions of finite support over non-deterministic processes for its semantics. We propose the notion of a stable distribution and show that every distribution can be unfolded into a stable distribution by chasing its (partial) $\tau$-transitions. Our framework, including the notion of branching probabilistic bisimulation, builds on that of [19, 16].

Another trait of the current paper, as in [19, 16], is that distributions are taken as semantic foundation for bisimilarity, rather than seeing bisimilarity primarily as an equivalence relation on non-deterministic processes, which is subsequently lifted to an equivalence relation on distributions, as is the case for the notion of branching probabilistic bisimilarity of [27, 26] and also of [2, 1]. The idea to consider distributions as first-class citizens for probabilistic bisimilarity stems from [11]. In the systematic overview of the spectrum [3], also Baier et al. argue that a behavioral relation on distributions is needed to properly deal with silent moves.

Metric spaces and complete metric spaces, as well as their associated categories, have various uses in concurrency theory. In the setting of semantics of probabilistic systems, metric topology has been advocated as underlying denotational domain, for example in [5, 21, 25]. For quantitative comparison of Markov systems, metrics and pseudo-metric have been proposed for a quantitative notion of behavior equivalence, see e.g. [10, 13, 7]. The specific use of metric topology in this paper to derive an existential property of a transition system seems new.

The remainder of the paper is organized as follows. Section 2 collects some definitions from metric topology and establishes some auxiliary results. A simple process language with non-deterministic and probabilistic choice is introduced in Section 3, together with examples and basic properties of the operational semantics. Our definition of branching probabilistic bisimilarity is given in Section 4, followed by a congruence result with respect to probabilistic composition and a confluence property. The main contribution of the paper is presented in Sections 5 and 6. Section 5 shows in a series of continuity lemmas that the set of branching probabilistic bisimilar descendants is a (sequentially) compact set. Section 6 exploits these results to argue that unfolding of a distribution by inert $\tau$-transitions has a stable end point, meaning that a stable branchingly equivalent distribution can be reached. With that result in place, a cancellation law for branching probabilistic bisimilarity is established. Finally, Section 7 wraps up with concluding remarks and a discussion of future work.

## 2 Preliminaries

For a non-empty set $X$, we define $\mathit{Distr}(X)$ as the set of all probability distributions over $X$ of finite support, i.e., $\mathit{Distr}(X) = \{\, \mu : X \to [0,1] \mid \sum_{x\in X} \mu(x) = 1,\ \mu(x) > 0 \text{ for finitely many } x \in X \,\}$. We use $\mathit{spt}(\mu)$ to denote the finite set $\{\, x \in X \mid \mu(x) > 0 \,\}$. Often, we write $\mu = \bigoplus_{i\in I} p_i \cdot x_i$ for an index set $I$, $p_i \geqslant 0$ and $x_i \in X$ for $i \in I$, where $p_i > 0$ for finitely many $i \in I$. Implicitly, we assume $\sum_{i\in I} p_i = 1$. We also write $r\mu \oplus (1-r)\nu$ and, equivalently, $\mu \,{}_r\!\oplus\, \nu$ for $\mu, \nu \in \mathit{Distr}(X)$ and $0 \leqslant r \leqslant 1$. As expected, we have that $(r\mu \oplus (1-r)\nu)(x) = (\mu \,{}_r\!\oplus\, \nu)(x) = r\mu(x) + (1-r)\nu(x)$ for $x \in X$. The *Dirac distribution* on $x$, the unique distribution with support $x$, is denoted $\delta(x)$.

The set $\mathit{Distr}(X)$ becomes a complete[1] metric space when endowed with the sup-norm [14], given by $d(\mu,\nu) = \sup_{x\in X} |\mu(x) - \nu(x)|$. This distance is also known as the distance of uniform convergence or Chebyshev distance.

**Theorem 1.** *If $Y \subseteq X$ is finite, then $\mathit{Distr}(Y)$ is a sequentially compact subspace of $\mathit{Distr}(X)$. This means that every sequence in $\mathit{Distr}(Y)$ has a convergent subsequence with a limit in $\mathit{Distr}(Y)$.*

*Proof.* $\mathit{Distr}(Y)$ is a bounded subset of $\mathbb{R}^n$, where $n := |Y|$ is the size of $Y$. It also is closed. For $\mathbb{R}^n$ equipped with the Euclidean metric, the sequential compactness of closed and bounded subsets is known as the Bolzano-Weierstrass theorem [23]. When using the Chebyshev metric, the same proof applies. □

In Section 5 we use the topological structure of the set of distributions over non-deterministic processes to study unfolding of partial $\tau$-transitions. There we make use of the following representation property.

**Lemma 2.** *Suppose the sequence of distributions $(\mu_i)_{i=0}^{\infty}$ converges to the distribution $\mu$ in $\mathit{Distr}(X)$. Then a sequence of distributions $(\mu'_i)_{i=0}^{\infty}$ in $\mathit{Distr}(X)$ and a sequence of probabilities $(r_i)_{i=0}^{\infty}$ in $[0,1]$ exist such that $\mu_i = (1-r_i)\,\mu \oplus r_i\,\mu'_i$ for $i \in \mathbb{N}$ and $\lim_{i\to\infty} r_i = 0$.*

*Proof.* Let $i \in \mathbb{N}$. For $x \in \mathit{spt}(\mu)$, the quotient $\mu_i(x)/\mu(x)$ is non-negative, but may exceed 1. However, $0 \leqslant \min\{\, \frac{\mu_i(x)}{\mu(x)} \mid x \in \mathit{spt}(\mu) \,\} \leqslant 1$, since the numerator cannot strictly exceed the denominator for all $x \in \mathit{spt}(\mu)$. Let $r_i = 1 - \min\{\, \frac{\mu_i(x)}{\mu(x)} \mid x \in \mathit{spt}(\mu) \,\}$ for $i \in \mathbb{N}$. Then we have $0 \leqslant r_i \leqslant 1$.

[1] A *Cauchy sequence* is a sequence of points in a metric space whose elements become arbitrarily close to each other as the sequence progresses. The space is *complete* if every such sequence has a limit within the space.

For $i \in \mathbb{N}$, define $\mu_i' \in \mathit{Distr}(X)$ as follows. If $r_i > 0$ then $\mu_i'(x) = 1/r_i \cdot \big[\mu_i(x) - (1-r_i)\mu(x)\big]$ for $x \in X$; if $r_i = 0$ then $\mu_i' = \mu$. We verify for $r_i > 0$ that $\mu_i'$ is indeed a distribution: (i) For $x \notin \mathit{spt}(\mu)$ it holds that $\mu(x) = 0$, and therefore $\mu_i'(x) = 1/r_i \cdot \mu_i(x) \geqslant 0$. For $x \in \mathit{spt}(\mu)$,

$$\mu_i'(x) = 1/r_i \cdot \big[\mu_i(x) - (1-r_i)\mu(x)\big] = \mu(x)/r_i \cdot \big[\frac{\mu_i(x)}{\mu(x)} - \frac{\mu_i(x_{min})}{\mu(x_{min})}\big] \geqslant 0$$

for $x_{min} \in \mathit{spt}(\mu)$ such that $\mu_i(x_{min})/\mu(x_{min})$ is minimal. (ii) In addition,

$$\begin{aligned}\sum\{\,\mu_i'(x) \mid x \in X\,\} &= 1/r_i \cdot \sum\{\,\mu_i(x) \mid x \notin \mathit{spt}(\mu)\,\} + 1/r_i \cdot \sum\{\,\mu_i(x) - (1-r_i)\mu(x) \mid x \in \mathit{spt}(\mu)\,\} = \\ & 1/r_i \cdot \sum\{\,\mu_i(x) \mid x \in X\,\} - (1-r_i)/r_i \cdot \sum\{\,\mu(x) \mid x \in \mathit{spt}(\mu)\,\} = 1/r_i - (1-r_i)/r_i = r_i/r_i = 1.\end{aligned}$$

Therefore, $0 \leqslant \mu_i'(x) \leqslant 1$ and $\sum\{\,\mu_i'(x) \mid x \in X\,\} = 1$.

Now we prove that $\mu_i = (1-r_i)\mu \oplus r_i\mu_i'$. If $r_i = 0$, then $\mu_i = \mu$, $\mu_i' = \mu$, and $\mu_i = (1-r_i)\mu \oplus r_i\mu_i'$. If $r_i > 0$, then $\mu_i(x) = (1-r_i)\mu(x) \oplus r_i\mu_i'(x)$ by definition of $\mu_i'(x)$ for all $x \in X$. Thus, also $\mu_i = (1-r_i)\mu \oplus r_i\mu_i'$ in this case.

Finally, we show that $\lim_{i\to\infty} r_i = 0$. Let $x_{min}' \in \mathit{spt}(\mu)$ be such that $\mu(x_{min}')$ is minimal. Then we have

$$r_i = 1 - \min\{\,\frac{\mu_i(x)}{\mu(x)} \mid x \in \mathit{spt}(\mu)\,\} = \max\{\,\frac{\mu(x) - \mu_i(x)}{\mu(x)} \mid x \in \mathit{spt}(\mu),\, \mu(x) \geqslant \mu_i(x)\,\} \leqslant \frac{d(\mu,\mu_i)}{\mu(x_{min}')}$$

By assumption, $\lim_{i\to\infty} d(\mu,\mu_i) = 0$. Hence also $\lim_{i\to\infty} r_i = 0$, as was to be shown. □

The following combinatorial result is helpful in the sequel.

**Lemma 3.** *Let $I$ and $J$ be finite index sets, $p_i, q_j \in [0,1]$ and $\mu_i, \nu_j \in \mathit{Distr}(X)$, for $i \in I$ and $j \in J$, such that $\bigoplus_{i\in I} p_i\mu_i = \bigoplus_{j\in J} q_j\nu_j$. Then $r_{ij} \geqslant 0$ and $\rho_{ij} \in \mathit{Distr}(X)$ exist such that $\sum_{j\in J} r_{ij} = p_i$ and $p_i \cdot \mu_i = \bigoplus_{j\in J} r_{ij} \cdot \rho_{ij}$ for all $i \in I$, and $\sum_{i\in I} r_{ij} = q_j$ and $q_j \cdot \nu_j = \bigoplus_{i\in I} r_{ij} \cdot \rho_{ij}$ for all $j \in J$.*

*Proof.* Let $\xi = \bigoplus_{i\in I} p_i \cdot \mu_i = \bigoplus_{j\in J} q_j \cdot \nu_j$. We define $r_{ij} = \sum_{x\in \mathit{spt}(\xi)} \dfrac{p_i\mu_i(x) \cdot q_j\nu_j(x)}{\xi(x)}$ for all $i \in I$ and $j \in J$. In case $r_{ij} = 0$, choose $\rho_{ij} \in \mathit{Distr}(X)$ arbitrarily. In case $r_{ij} \neq 0$, define $\rho_{ij} \in \mathit{Distr}(X)$, for $i \in I$ and $j \in J$, by

$$\rho_{ij}(x) = \begin{cases} \dfrac{p_i\mu_i(x) \cdot q_j\nu_j(x)}{r_{ij}\,\xi(x)} & \text{if } \xi(x) > 0, \\ 0 & \text{otherwise} \end{cases}$$

for all $x \in X$. By definition of $r_{ij}$ and $\rho_{ij}$ it holds that $\sum\{\,\rho_{ij}(x) \mid x \in X\,\} = 1$. So, $\rho_{ij} \in \mathit{Distr}(X)$ indeed.

We verify $\sum_{j\in J} r_{ij} = p_i$ and $p_i \cdot \mu_i = \bigoplus_{j\in J} r_{ij} \cdot \rho_{ij}$ for $i \in I$.

$$\begin{aligned}\textstyle\sum_{j\in J} r_{ij} &= \textstyle\sum_{j\in J} \sum_{x\in \mathit{spt}(\xi)} p_i\mu_i(x) \cdot q_j\nu_j(x)/\xi(x) \\ &= \textstyle\sum_{x\in \mathit{spt}(\xi)} p_i\mu_i(x) \cdot \sum_{j\in J} q_j\nu_j(x)/\xi(x) \\ &= \textstyle\sum_{x\in \mathit{spt}(\xi)} p_i\mu_i(x) && (\text{since } \xi = \textstyle\bigoplus_{j\in J} q_j \cdot \nu_j) \\ &= p_i \textstyle\sum_{x\in \mathit{spt}(\xi)} \mu_i(x) \\ &= p_i\,.\end{aligned}$$

Next, pick $y \in X$ and $i \in I$. If $\xi(y) = 0$, then $p_i\mu_i(y) = 0$, since $\xi(y) = \sum_{i\in I} p_i\mu_i(y)$, and $r_{ij} = 0$ or $\rho_{ij}(y) = 0$ for all $j \in J$, by the various definitions, thus $\sum_{j\in J} r_{ij}\rho_{ij}(y) = 0$ as well.

Suppose $\xi(y) > 0$. Put $J_i = \{\, j \in J \mid r_{ij} > 0 \,\}$. If $j \in J \backslash J_i$, i.e. if $r_{ij} = 0$, then $p_i \mu_i(y) q_j \nu_j(y)/\xi(y) = 0$ by definition of $r_{ij}$. Therefore we have

$$\begin{aligned}
\textstyle\sum_{j\in J} r_{ij}\rho_{ij}(y) &= \textstyle\sum_{j\in J_i} r_{ij}\rho_{ij}(y) \\
&= \textstyle\sum_{j\in J_i} r_{ij} p_i \mu_i(y) \cdot q_j \nu_j(y)/(r_{ij}\xi(y)) \\
&= \textstyle\sum_{j\in J_i} p_i \mu_i(y) \cdot q_j \nu_j(y)/\xi(y) \\
&= \textstyle\sum_{j\in J} p_i \mu_i(y) \cdot q_j \nu_j(y)/\xi(y) && \text{(summand zero for } j \in J\backslash J_i\text{)} \\
&= p_i \mu_i(y)/\xi(y) \cdot \textstyle\sum_{j\in J} q_j \nu_j(y) \\
&= p_i \mu_i(y) && \text{(since } \xi = \textstyle\bigoplus_{j\in J} q_j \cdot \nu_j\text{)}.
\end{aligned}$$

The statements $\sum_{i\in I} r_{ij} = q_j$ and $q_j \cdot \nu_j = \bigoplus_{i\in I} r_{ij} \cdot \rho_{ij}$ for $j \in J$ follow by symmetry. □

## 3 An elementary processes language

In this section we define a syntax and transition system semantics for non-deterministic and probabilistic processes. Depending on the top operator, following [6], a process is either a non-deterministic process $E \in \mathscr{E}$, with constant $\mathbf{0}$, prefix operators $\alpha\cdot$ and non-deterministic choice $+$, or a probabilistic process $P \in \mathscr{P}$, with the Dirac operator $\partial$ and probabilistic choices ${}_r\oplus$.

**Definition 4** (Syntax)**.** *The classes $\mathscr{E}$ and $\mathscr{P}$ of non-deterministic and probabilistic processes, respectively, over the set of actions $\mathscr{A}$, are given by*

$$E ::= \mathbf{0} \mid \alpha\cdot P \mid E + E \qquad\qquad P ::= \partial(E) \mid P \,{}_r\oplus\, P$$

*with actions $\alpha$ from $\mathscr{A}$ and where $0 \leqslant r \leqslant 1$.*

We use $E, F, \ldots$ to range over $\mathscr{E}$ and $P, Q, \ldots$ to range over $\mathscr{P}$. The probabilistic process $P_1 \,{}_r\oplus\, P_2$ behaves as $P_1$ with probability $r$ and behaves as $P_2$ with probability $1 - r$.

We introduce a complexity measure $c : \mathscr{E} \cup \mathscr{P} \to \mathbb{N}$ for non-deterministic and probabilistic processes based on the size of a process. It is given by $c(\mathbf{0}) = 0$, $c(a\cdot P) = c(P) + 1$, $c(E + F) = c(E) + c(F)$, and $c(\partial(E)) = c(E) + 1$, $c(P \,{}_r\oplus\, Q) = c(P) + c(Q)$.

**Examples** As illustration, we provide the following pairs of non-deterministic processes, which are branching probabilistic bisimilar in the sense of Definition 9.

(i) $\mathbf{H_1} = a\cdot\big(P \,{}_{\frac14}\oplus\, (P \,{}_{\frac13}\oplus\, Q)\big)$ and $\mathbf{H_2} = a\cdot\big(P \,{}_{\frac12}\oplus\, (Q \,{}_{\frac12}\oplus\, Q)\big)$

(ii) $\mathbf{G_1} = a\cdot(P \,{}_{\frac12}\oplus\, Q)$ and $\mathbf{G_2} = a\cdot\big(\partial\big(\tau\cdot(P \,{}_{\frac12}\oplus\, Q)\big) \,{}_{\frac13}\oplus\, (P \,{}_{\frac12}\oplus\, Q)\big)$

(iii) $\mathbf{I_1} = a\cdot\partial(b\cdot P + \tau\cdot Q)$ and $\mathbf{I_2} = a\cdot\partial(\tau\cdot\partial(b\cdot P + \tau\cdot Q) + b\cdot P + \tau\cdot Q)$

The examples $\mathbf{H_1}$ and $\mathbf{H_2}$ are taken from [22], and $\mathbf{G_1}$ and $\mathbf{G_2}$ are taken from [16]. The processes $\mathbf{G_2}$ and $\mathbf{I_2}$ contain a so-called inert $\tau$-transition.

As usual, the SOS semantics for $\mathscr{E}$ and $\mathscr{P}$ makes use of two types of transition relations [20, 6, 16].

**Definition 5** (Operational semantics)**.**
*(a) The transition relations* $\to \subseteq \mathscr{E} \times \mathscr{A} \times Distr(\mathscr{E})$ *and* $\mapsto \subseteq \mathscr{P} \times Distr(\mathscr{E})$ *are given by*

$$\frac{P \mapsto \mu}{\alpha \cdot P \xrightarrow{\alpha} \mu}\,(\text{PREF})$$

$$\frac{E_1 \xrightarrow{\alpha} \mu_1}{E_1 + E_2 \xrightarrow{\alpha} \mu_1}\,(\text{ND-CHOICE 1}) \qquad \frac{E_2 \xrightarrow{\alpha} \mu_2}{E_1 + E_2 \xrightarrow{\alpha} \mu_2}\,(\text{ND-CHOICE 2})$$

$$\frac{}{\partial(E) \mapsto \delta(E)}\,(\text{DIRAC}) \qquad \frac{P_1 \mapsto \mu_1 \quad P_2 \mapsto \mu_2}{P_1 \,{}_r\!\oplus\, P_2 \mapsto \mu_1 \,{}_r\!\oplus\, \mu_2}\,(\text{P-CHOICE})$$

*(b) The transition relation* $\to \subseteq Distr(\mathscr{E}) \times \mathscr{A} \times Distr(\mathscr{E})$ *is such that* $\mu \xrightarrow{\alpha} \mu'$ *whenever* $\mu = \bigoplus_{i\in I} p_i \cdot E_i$, $\mu' = \bigoplus_{i\in I} p_i \cdot \mu_i'$, *and* $E_i \xrightarrow{\alpha} \mu_i'$ *for all* $i \in I$.

In rule (DIRAC) of the relation $\mapsto$ we have that the syntactic Dirac process $\partial(E)$ is coupled to the semantic Dirac distribution $\delta(E)$. Similarly, in (P-CHOICE), the syntactic probabilistic operator ${}_r\!\oplus$ in $P_1 \,{}_r\!\oplus\, P_2$ is replaced by semantic probabilistic composition in $\mu_1 \,{}_r\!\oplus\, \mu_2$. Thus, with each probabilistic process $P \in \mathscr{P}$ we associate a distribution $[\![P]\!] \in Distr(\mathscr{E})$ as follows: $[\![\partial(E)]\!] = \delta(E)$ and $[\![P \,{}_r\!\oplus\, Q]\!] = [\![P]\!] \,{}_r\!\oplus\, [\![Q]\!]$, which is the distribution $r[\![P]\!] \oplus (1-r)[\![Q]\!]$.

The relation $\to$ for non-deterministic processes is finitely branching, but the relation $\to$ for probabilistic processes is not. Following [27, 26], the transition relation $\to$ on distributions as given by Definition 5 allows for a probabilistic combination of non-deterministic alternatives resulting in a so-called combined transition. For example, for the process $E = a\cdot(P \,{}_{\frac{1}{2}}\!\oplus\, Q) + a\cdot(P \,{}_{\frac{1}{3}}\!\oplus\, Q)$ of [6], we have that the Dirac process $\partial(E) = \partial(a\cdot(P \,{}_{\frac{1}{2}}\!\oplus\, Q) + a\cdot(P \,{}_{\frac{1}{3}}\!\oplus\, Q))$ provides an $a$-transition to $[\![P \,{}_{\frac{1}{2}}\!\oplus\, Q]\!]$ as well as an $a$-transition to $[\![P \,{}_{\frac{1}{3}}\!\oplus\, Q]\!]$. So, since we can represent the distribution $\delta(E)$ by $\delta(E) = \frac{1}{2}\delta(E) \oplus \frac{1}{2}\delta(E)$, the distribution $\delta(E)$ also has a combined transition

$$\delta(E) = \tfrac{1}{2}\delta(E) \oplus \tfrac{1}{2}\delta(E) \xrightarrow{a} \tfrac{1}{2}[\![P \,{}_{\frac{1}{2}}\!\oplus\, Q]\!] \oplus \tfrac{1}{2}[\![P \,{}_{\frac{1}{3}}\!\oplus\, Q]\!] = [\![P \,{}_{\frac{5}{12}}\!\oplus\, Q]\!].$$

As noted in [28], the ability to combine transitions is crucial for obtaining transitivity of probabilistic process equivalences that take internal actions into account.

**Example** Referring to the examples of processes above, we have, e.g,

$$\begin{aligned}
\mathbf{H_1}\colon &\quad \delta(a\cdot(P \,{}_{\frac{1}{4}}\!\oplus\, (P \,{}_{\frac{1}{3}}\!\oplus\, Q))) \xrightarrow{a} [\![P \,{}_{\frac{1}{4}}\!\oplus\, (P \,{}_{\frac{1}{3}}\!\oplus\, Q)]\!] = \tfrac{1}{2}[\![P]\!] \oplus \tfrac{1}{2}[\![Q]\!] \\
\mathbf{H_2}\colon &\quad \delta(a\cdot(P \,{}_{\frac{1}{2}}\!\oplus\, (Q \,{}_{\frac{1}{2}}\!\oplus\, Q))) \xrightarrow{a} [\![P \,{}_{\frac{1}{2}}\!\oplus\, (Q \,{}_{\frac{1}{2}}\!\oplus\, Q)]\!] = \tfrac{1}{2}[\![P]\!] \oplus \tfrac{1}{2}[\![Q]\!] \\
\mathbf{G_2}\colon &\quad a\cdot\big(\partial\big(\tau\cdot(P \,{}_{\frac{1}{2}}\!\oplus\, Q)\big) \,{}_{\frac{1}{3}}\!\oplus\, (P \,{}_{\frac{1}{2}}\!\oplus\, Q)\big) \xrightarrow{a} \delta\big(\tau\cdot(P \,{}_{\frac{1}{2}}\!\oplus\, Q)\big) \,{}_{\frac{1}{3}}\!\oplus\, (P \,{}_{\frac{1}{2}}\!\oplus\, Q).
\end{aligned}$$

Because a transition of a probabilistic process yields a distribution, the $a$-transitions of $\mathbf{H_1}$ and $\mathbf{H_2}$ have the same target. It is noted that $\mathbf{G_2}$ doesn't provide a further transition unless both its components $P$ and $Q$ do so to match the transition of $\tau\cdot(P \,{}_{\frac{1}{2}}\!\oplus\, Q)$.

In preparation to the definition of the notion of branching probabilistic bisimilarity in Section 4 we introduce some notation.

**Definition 6.** *For* $\mu, \mu' \in Distr(\mathscr{E})$ *and* $\alpha \in \mathscr{A}$ *we write* $\mu \xrightarrow{(\alpha)} \mu'$ *iff (i)* $\mu \xrightarrow{\alpha} \mu'$, *or (ii)* $\alpha = \tau$ *and* $\mu' = \mu$, *or (iii)* $\alpha = \tau$ *and there exist* $\mu_1, \mu_2, \mu_1', \mu_2' \in Distr(\mathscr{E})$ *such that* $\mu = \mu_1 \,{}_r\!\oplus\, \mu_2$, $\mu' = \mu_1' \,{}_r\!\oplus\, \mu_2'$, $\mu_1 \xrightarrow{\tau} \mu_1'$ *and* $\mu_2 = \mu_2'$ *for some* $r \in (0,1)$.

Cases (i) and (ii) in the definition above correspond with the limits $r = 1$ and $r = 0$ of case (iii). We use $\Rightarrow$ to denote the reflexive transitive closure of $\xrightarrow{(\tau)}$. A transition $\mu \xrightarrow{(\tau)} \mu'$ is called a partial transition, and a transition $\mu \Rightarrow \mu'$ is called a weak transition.

**Example**

(a) According to Definition 6 we have

$$\tfrac{1}{3}\delta(\tau\cdot(P\,{}_{\frac{1}{2}}\!\oplus Q))\oplus\tfrac{2}{3}[\![P\,{}_{\frac{1}{2}}\!\oplus Q]\!] \xrightarrow{(\tau)} \tfrac{1}{3}[\![P\,{}_{\frac{1}{2}}\!\oplus Q]\!]\oplus\tfrac{2}{3}[\![P\,{}_{\frac{1}{2}}\!\oplus Q]\!] = [\![P\,{}_{\frac{1}{2}}\!\oplus Q]\!].$$

(b) There are typically multiple ways to construct a weak transition $\Rightarrow$. Consider the weak transition $\frac{1}{2}\delta(\tau\cdot\partial(\tau\cdot P))\oplus\frac{1}{3}\delta(\tau\cdot P)\oplus\frac{1}{6}[\![P]\!]\Rightarrow[\![P]\!]$ which can be obtained, among uncountably many other possibilities, via

$$\tfrac{1}{2}\delta(\tau\cdot\partial(\tau\cdot P))\oplus\tfrac{1}{3}\delta(\tau\cdot P)\oplus\tfrac{1}{6}[\![P]\!] \xrightarrow{(\tau)} \tfrac{1}{2}\delta(\tau\cdot P))\oplus\tfrac{1}{3}\delta(\tau\cdot P)\oplus\tfrac{1}{6}[\![P]\!] = \tfrac{5}{6}\delta(\tau\cdot P)\oplus\tfrac{1}{6}[\![P]\!] \xrightarrow{(\tau)} [\![P]\!],$$

or via

$$\tfrac{1}{2}\delta(\tau\cdot\partial(\tau\cdot P))\oplus\tfrac{1}{3}\delta(\tau\cdot P)\oplus\tfrac{1}{6}[\![P]\!] \xrightarrow{(\tau)} \tfrac{1}{2}\delta(\tau\cdot\partial(\tau\cdot P))\oplus\tfrac{1}{3}\delta(P)\oplus\tfrac{1}{6}[\![P]\!] = \tfrac{1}{2}\delta(\tau\cdot\partial(\tau\cdot P))\oplus\tfrac{1}{2}\delta(P) \xrightarrow{(\tau)} \tfrac{1}{2}\delta(\tau\cdot P)\oplus\tfrac{1}{2}[\![P]\!] \xrightarrow{(\tau)} \tfrac{1}{2}[\![P]\!]\oplus\tfrac{1}{2}[\![P]\!] = [\![P]\!].$$

(c) The distribution $\frac{1}{2}\delta(\tau\cdot\partial(a\cdot\partial(\mathbf{0})+b\cdot\partial(\mathbf{0})))\oplus\frac{1}{2}\delta(a\cdot\partial(c\cdot\partial(\mathbf{0})))$ doesn't admit a $\tau$-transition nor an $a$-transition. However, we have

$$\tfrac{1}{2}\delta(\tau\cdot\partial(a\cdot\partial(\mathbf{0})+b\cdot\partial(\mathbf{0})))\oplus\tfrac{1}{2}\delta(a\cdot\partial(c\cdot\partial(\mathbf{0}))) \xrightarrow{(\tau)} \tfrac{1}{2}\partial(a\cdot\partial(\mathbf{0})+b\cdot\partial(\mathbf{0}))\oplus\tfrac{1}{2}\delta(a\cdot\partial(c\cdot\partial(\mathbf{0}))) \xrightarrow{a} \tfrac{1}{2}\delta(\mathbf{0})\oplus\tfrac{1}{2}\delta(c\cdot\partial(\mathbf{0})).$$

The following lemma states that the transitions $\xrightarrow{\alpha}$, $\xrightarrow{(\alpha)}$, and $\Rightarrow$ of Definitions 5 and 6 can be probabilistically composed.

**Lemma 7.** *Let, for a finite index set $I$, $\mu_i,\mu_i'\in Distr(\mathscr{E})$ and $p_i\geqslant 0$ such that $\sum_{i\in I}p_i=1$.*

*(a) If $\mu_i\xrightarrow{\alpha}\mu_i'$ for all $i\in I$, then $\bigoplus_{i\in I}p_i\cdot\mu_i\xrightarrow{\alpha}\bigoplus_{i\in I}p_i\cdot\mu_i'$.*

*(b) If $\mu_i\xrightarrow{(\tau)}\mu_i'$ for all $i\in I$, then $\bigoplus_{i\in I}p_i\cdot\mu_i\xrightarrow{(\tau)}\bigoplus_{i\in I}p_i\cdot\mu_i'$.*

*(c) If $\mu_i\Rightarrow\mu_i'$ for all $i\in I$, then $\bigoplus_{i\in I}p_i\cdot\mu_i\Rightarrow\bigoplus_{i\in I}p_i\cdot\mu_i'$.*

*Proof.* Let $\mu=\bigoplus_{i\in I}p_i\cdot\mu_i$ and $\mu'=\bigoplus_{i\in I}p_i\cdot\mu_i'$. Without loss of generality, we may assume that $p_i>0$ for all $i\in I$.

(a) Suppose $\mu_i\xrightarrow{\alpha}\mu_i'$ for all $i\in I$. Then, by Definition 5, $\mu_i=\bigoplus_{j\in J_i}p_{ij}\cdot E_{ij}$, $\mu_i'=\bigoplus_{j\in J_i}p_{ij}\cdot\eta_{ij}$, and $E_{ij}\xrightarrow{\alpha}\eta_{ij}$ for $j\in J_i$ for a suitable index set $J_i$, $p_{ij}>0$ and $\eta_{ij}\in Distr(\mathscr{E})$. Define the index set $K$ and probabilities $q_k$ for $k\in K$ by $K=\{\,(i,j)\mid i\in I,\,j\in J_i\,\}$ and $q_{(i,j)}=p_ip_{ij}$ for $(i,j)\in K$, so that $\sum_{k\in K}q_k=1$. Then we have $\mu=\bigoplus_{k\in K}q_k\cdot E_{ij}$ and $\mu'=\bigoplus_{k\in K}q_k\cdot\eta_{ij}$. Therefore, by Definition 5, it follows that $\mu\xrightarrow{\alpha}\mu'$.

(b) Let $\mu_i\xrightarrow{(\tau)}\mu_i'$ for all $i\in I$. Then, for all $i\in I$, by Definition 6, there exists $r_i\in[0,1]$ and $\mu_i^{\rm stay},\mu_i^{\rm go},\mu_i''\in Distr(\mathscr{E})$, such that $\mu_i=\mu_i^{\rm stay}\,{}_{r_i}\!\oplus\mu_i^{\rm go}$, $\mu_i'=\mu_i^{\rm stay}\,{}_{r_i}\!\oplus\mu_i''$, and either $r_i=1$ or $\mu_i^{\rm go}\xrightarrow{\tau}\mu_i''$. In case $r_i=0$ for all $i\in I$, we have that $\mu_i\xrightarrow{\tau}\mu_i'$ for all $i\in I$, and thus $\mu\xrightarrow{\tau}\mu'$ by the first claim of the lemma, and $\mu\xrightarrow{(\tau)}\mu'$ by Definition 6(i). In case $r_i=1$ for all $i\in I$, we have $\mu'=\mu$ and thus $\mu\xrightarrow{(\tau)}\mu'$ by Definition 6(ii). Otherwise, let $I':=\{\,i\in I\mid r_i<1\,\}$, $r=\sum_{i\in I}p_i\cdot r_i$, $\mu^{\rm stay}:=\bigoplus_{i\in I}\frac{p_i\cdot r_i}{r}\cdot\mu_i^{\rm stay}$, $\mu^{\rm go}:=\bigoplus_{i\in I'}\frac{p_i\cdot(1-r_i)}{1-r}\cdot\mu_i^{\rm go}$ and $\mu'':=\bigoplus_{i\in I'}\frac{p_i\cdot(1-r_i)}{1-r}\cdot\mu_i''$. Then $\mu^{\rm go}\xrightarrow{\tau}\mu''$ by the first claim of the lemma. Moreover, $\mu=\mu^{\rm stay}\,{}_{r}\!\oplus\mu^{\rm go}$, $\mu'=\mu^{\rm stay}\,{}_{r}\!\oplus\mu''$ and $r\in(0,1)$. So $\mu\xrightarrow{(\tau)}\mu'$ by Definition 6(iii).

(c) Let $\mu_i\Rightarrow\mu_i'$ for all $i\in I$. As $I$ is finite and $\Rightarrow$ is reflexive, there exists an $n\in\mathbb{N}$ such that $\mu_i=\mu_i^{(0)}\xrightarrow{(\tau)}\mu_i^{(1)}\xrightarrow{(\tau)}\ldots\xrightarrow{(\tau)}\mu_i^{(n)}=\mu_i'$ for all $i\in I$. Now $\mu\Rightarrow\mu'$ follows by $n$ applications of the second statement of the lemma. □

Likewise, the next lemma allows *probabilistic decomposition* of transitions $\xrightarrow{\alpha}$, $\xrightarrow{(\alpha)}$ and $\Rightarrow$.

**Lemma 8.** *Let $\mu, \mu' \in Distr(\mathscr{E})$ and $\mu = \bigoplus_{i\in I} p_i \cdot \mu_i$ with $p_i > 0$ for $i \in I$.*

*(a) If $\mu \xrightarrow{\alpha} \mu'$, then there are $\mu'_i$ for $i \in I$ such that $\mu_i \xrightarrow{\alpha} \mu'_i$ for $i \in I$ and $\mu' = \bigoplus_{i\in I} p_i \cdot \mu'_i$.*

*(b) If $\mu \xrightarrow{(\tau)} \mu'$, then there are $\mu'_i$ for $i \in I$ such that $\mu_i \xrightarrow{(\tau)} \mu'_i$ for $i \in I$ and $\mu' = \bigoplus_{i\in I} p_i \cdot \mu'_i$.*

*(c) If $\mu \Rightarrow \mu'$, then there are $\mu'_i$ for $i \in I$ such that $\mu_i \Rightarrow \mu'_i$ for $i \in I$ and $\mu' = \bigoplus_{i\in I} p_i \cdot \mu'_i$.*

*Proof.* (a) Suppose $\mu \xrightarrow{\alpha} \mu'$. By Definition 5 $\mu = \bigoplus_{j\in J} q_j \cdot E_j$, $\mu' = \bigoplus_{j\in J} q_j \cdot \eta_j$, and $E_j \xrightarrow{\alpha} \eta_j$ for all $j \in J$, for suitable index set $J$, $q_j > 0$, $E_j \in \mathscr{E}$, and $\eta_j \in Distr(\mathscr{E})$. By Lemma 3 there are $r_{ij} \geqslant 0$ and $\rho_{ij} \in Distr(\mathscr{E})$ such that $\sum_{j\in J} r_{ij} = p_i$ and $p_i\mu_i = \bigoplus_{j\in J} r_{ij}\rho_{ij}$ for $i \in I$, and $\sum_{i\in I} r_{ij} = q_j$ and $q_j \cdot \delta(E_j) = \bigoplus_{i\in I} r_{ij}\rho_{ij}$ for all $j \in J$. Hence, $\rho_{ij} = \delta(E_j)$ for $i \in I$, $j \in J$.

For all $i \in I$, let $\mu'_i = \bigoplus_{j\in J} (r_{ij}/p_i)\eta_j$. Then $\mu_i \xrightarrow{\alpha} \mu'_i$, for all $i \in I$, by Lemma 7(a). Moreover, it holds that $\bigoplus_{i\in I} p_i\mu'_i = \bigoplus_{i\in I} p_i \cdot \bigoplus_{j\in J} (r_{ij}/p_i)\eta_j = \bigoplus_{j\in J} \bigoplus_{i\in I} r_{ij} \cdot \eta_j = \bigoplus_{j\in J} q_j \cdot \eta_j = \mu'$.

(b) Suppose $\mu \xrightarrow{(\tau)} \mu'$. By Definition 6, either (i) $\mu \xrightarrow{\tau} \mu'$, or (ii) $\mu' = \mu$, or (iii) there exist $\nu_1, \nu_2, \nu'_1, \nu'_2 \in Distr(\mathscr{E})$ such that $\mu = \nu_1 \,{}_r\!\oplus\, \nu_2$, $\mu' = \nu'_1 \,{}_r\!\oplus\, \nu'_2$, $\nu_1 \xrightarrow{\tau} \nu'_1$ and $\nu_2 = \nu'_2$ for some $r \in (0,1)$. In case (i), the required $\mu'_i$ exist by the first statement of this lemma. In case (ii) one can simply take $\mu'_i := \mu_i$ for all $i \in I$. Hence assume that case (iii) applies. Let $J := \{1,2\}$, $q_1 := r$ and $q_2 := 1 - r$. By Lemma 3 there are $r_{ij} \in [0,1]$ and $\rho_{ij} \in Distr(\mathscr{E})$ with $\sum_{j\in J} r_{ij} = p_i$ and $\mu_i = \bigoplus_{j\in J} \frac{r_{ij}}{p_i} \cdot \rho_{ij}$ for all $i \in I$, and $\sum_{i\in I} r_{ij} = q_j$ and $\nu_j = \bigoplus_{i\in I} \frac{r_{ij}}{q_j} \cdot \rho_{ij}$ for all $j \in J$.

Let $I' := \{i \in I \mid r_{i1} > 0\}$. Since $\nu_1 = \bigoplus_{i\in I'} \frac{r_{i1}}{r} \cdot \rho_{i1} \xrightarrow{\tau} \nu'_1$, by the first statement of the lemma, for all $i \in I'$ there are $\rho'_{i1}$ such that $\rho_{i1} \xrightarrow{\tau} \rho'_{i1}$ and $\nu'_1 = \bigoplus_{i\in I'} \frac{r_{i1}}{r} \cdot \rho'_{i1}$. For all $i \in I\backslash I'$ pick $\rho'_{i1} \in Distr(\mathscr{E})$ arbitrarily. It follows that $\mu_i = \rho_{i1} \,{}_{\frac{r_{i1}}{p_i}}\!\oplus\, \rho_{i2} \xrightarrow{(\tau)} \rho'_{i1} \,{}_{\frac{r_{i1}}{p_i}}\!\oplus\, \rho_{i2} =: \mu'_i$ for all $i \in I$. Moreover,

$$\textstyle\bigoplus_{i\in I} p_i \cdot \mu'_i = \bigoplus_{i\in I} p_i \cdot (\rho'_{i1} \,{}_{\frac{r_{i1}}{p_i}}\!\oplus\, \rho_{i2}) = (\bigoplus_{i\in I} \frac{r_{i1}}{r} \cdot \rho'_{i1}) \,{}_r\!\oplus\, (\bigoplus_{i\in I} \frac{r_{i2}}{1-r} \cdot \rho_{i2}) = \nu'_1 \,{}_r\!\oplus\, \nu_2 = \mu' \,.$$

(c) The last statement follows by transitivity from the second one. □

# 4 Branching probabilistic bisimilarity

In this section we recall the notion of branching probabilistic bisimilarity [16]. The notion is based on a decomposability property due to [9] and a transfer property.

**Definition 9** (Branching probabilistic bisimilarity)**.**

*(a) A relation $\mathscr{R} \subseteq Distr(\mathscr{E}) \times Distr(\mathscr{E})$ is called weakly decomposable iff it is symmetric and for all $\mu, \nu \in Distr(\mathscr{E})$ such that $\mu \mathscr{R} \nu$ and $\mu = \bigoplus_{i\in I} p_i \cdot \mu_i$ there are $\bar{\nu}, \nu_i \in Distr(\mathscr{E})$, for $i \in I$, such that*

$$\nu \Rightarrow \bar{\nu},\ \mu \mathscr{R} \bar{\nu},\ \bar{\nu} = \textstyle\bigoplus_{i\in I} p_i \cdot \nu_i,\ \text{and}\ \ \mu_i \mathscr{R} \nu_i \ \text{for all } i \in I.$$

*(b) A relation $\mathscr{R} \subseteq Distr(\mathscr{E}) \times Distr(\mathscr{E})$ is called a* branching *probabilistic bisimulation relation iff it is weakly decomposable and for all $\mu, \nu \in Distr(\mathscr{E})$ with $\mu \mathscr{R} \nu$ and $\mu \xrightarrow{\alpha} \mu'$, there are $\bar{\nu}, \nu' \in Distr(\mathscr{E})$ such that*

$$\nu \Rightarrow \bar{\nu},\ \bar{\nu} \xrightarrow{(\alpha)} \nu',\ \mu \mathscr{R} \bar{\nu},\ \text{and}\ \ \mu' \mathscr{R} \nu'.$$

*(c) Branching probabilistic bisimilarity $\leftrightarrows_b \subseteq Distr(\mathscr{E}) \times Distr(\mathscr{E})$ is defined as the largest branching probabilistic bisimulation relation on $Distr(\mathscr{E})$.*

Note that branching probabilistic bisimilarity is well-defined following the usual argument that any union of branching probabilistic bisimulation relations is again a branching probabilistic bisimulation relation. In particular, (weak) decomposability is preserved under arbitrary unions. As observed in [15], branching probabilistic bisimilarity is an equivalence relation.

Two non-deterministic processes are considered to be branching probabilistic bisimilar iff their Dirac distributions are, i.e., for $E,F \in \mathscr{E}$ we have $E \leftrightarrows_b F$ iff $\delta(E) \leftrightarrows_b \delta(F)$. Two probabilistic processes are considered to be branching probabilistic bisimilar iff their associated distributions over $\mathscr{E}$ are, i.e., for $P,Q \in \mathscr{P}$ we have $P \leftrightarrows_b Q$ iff $[\![P]\!] \leftrightarrows_b [\![Q]\!]$.

For a set $M \subseteq Distr(\mathscr{E})$, the convex closure $cc(M)$ is defined by

$$cc(M) = \{\textstyle\bigoplus_{i\in I} p_i\mu_i \mid \sum_{i\in I} p_i = 1,\ \mu_i \in M,\ I \text{ a finite index set}\}.$$

For a relation $\mathscr{R} \subseteq Distr(\mathscr{E}) \times Distr(\mathscr{E})$ the convex closure of $\mathscr{R}$ is defined by

$$cc(\mathscr{R}) = \{\, \langle \textstyle\bigoplus_{i\in I} p_i\mu_i, \bigoplus_{i\in I} p_i\nu_i\rangle \mid \mu_i \mathscr{R} \nu_i,\ \sum_{i\in I} p_i = 1,\ I \text{ a finite index set}\,\}.$$

The notion of weak decomposability has been adopted from [22, 24]. The underlying idea stems from [9]. Weak decomposability provides a convenient dexterity to deal with combined transitions as well as with sub-distributions. For example, regarding sub-distributions, to distinguish the probabilistic process $\frac{1}{2}\partial(a\cdot\partial(\mathbf{0})) \oplus \frac{1}{2}\partial(b\cdot\partial(\mathbf{0}))$ from $\partial(\mathbf{0})$ a branching probabilistic bisimulation relation relating $\frac{1}{2}\delta(a\cdot\partial(\mathbf{0})) \oplus \frac{1}{2}\delta(b\cdot\partial(\mathbf{0}))$ and $\delta(\mathbf{0})$ is by weak decomposability also required to relate $\delta(a\cdot\partial(\mathbf{0}))$ and $\delta(b\cdot\partial(\mathbf{0}))$ to subdistributions of a weak descendant of $\delta(\mathbf{0})$, which can only be $\delta(\mathbf{0})$ itself. Since $\delta(a\cdot\partial(\mathbf{0}))$ has an $a$-transition while $\delta(\mathbf{0})$ has not, and similar for a $b$-transition of $\delta(b\cdot\partial(\mathbf{0}))$, it follows that $\frac{1}{2}\partial(a\cdot\partial(\mathbf{0})) \oplus \frac{1}{2}\partial(b\cdot\partial(\mathbf{0}))$ and $\partial(\mathbf{0})$ are not branching probabilistic bisimilar.

By comparison, on finite processes, as used in this paper, the notion of branching probabilistic bisimilarity of Segala & Lynch [27] can be defined in our framework exactly as in (b) and (c) above, but taking a decomposable instead of a weakly decomposable relation, i.e. if $\mu \mathscr{R} \nu$ and $\mu = \bigoplus_{i\in I} p_i\mu_i$ then there are $\nu_i$ for $i \in I$ such that $\nu = \bigoplus_{i\in I} p_i\nu_i$ and $\mu_i \mathscr{R} \nu_i$ for $i \in I$. This yields a strictly finer equivalence.

**Example**

(a) The distributions $\delta(\mathbf{G_1}) = \delta(a\cdot(P \,{}_{\frac{1}{2}}\!\oplus Q))$ and $\delta(\mathbf{G_2}) = \delta(a\cdot(\partial(\tau\cdot(P \,{}_{\frac{1}{2}}\!\oplus Q)) \,{}_{\frac{1}{3}}\!\oplus (P \,{}_{\frac{1}{2}}\!\oplus Q)))$ both admit at the top level an $a$-transition only:

$$\delta(a\cdot(P \,{}_{\frac{1}{2}}\!\oplus Q)) \xrightarrow{a} \tfrac{1}{2}[\![P]\!] \oplus \tfrac{1}{2}[\![Q]\!]$$
$$\delta(a\cdot(\partial(\tau\cdot(P \,{}_{\frac{1}{2}}\!\oplus Q)) \,{}_{\frac{1}{3}}\!\oplus (P \,{}_{\frac{1}{2}}\!\oplus Q))) \xrightarrow{a} \tfrac{1}{3}\delta(\tau\cdot(P \,{}_{\frac{1}{2}}\!\oplus Q)) \oplus \tfrac{1}{3}[\![P]\!] \oplus \tfrac{1}{3}[\![Q]\!].$$

Let the relation $\mathscr{R}$ contain the pairs

$$\langle \delta(\tau\cdot(P \,{}_{\frac{1}{2}}\!\oplus Q)), \tfrac{1}{2}[\![P]\!] \oplus \tfrac{1}{2}[\![Q]\!]\rangle \quad \text{and} \quad \langle \mu,\mu\rangle \text{ for } \mu \in Distr(\mathscr{E}).$$

The symmetric closure $\mathscr{R}^\dagger$ of $\mathscr{R}$ is clearly a branching probabilistic bisimulation relation. We claim that therefore also its convex closure $cc(\mathscr{R}^\dagger)$ is a branching probabilistic bisimulation relation. Considering that $\langle \delta(\tau\cdot(P \,{}_{\frac{1}{2}}\!\oplus Q)), \frac{1}{2}[\![P]\!] \oplus \frac{1}{2}[\![Q]\!]\rangle$ and $\langle \frac{1}{2}[\![P]\!] \oplus \frac{1}{2}[\![Q]\!], \frac{1}{2}[\![P]\!] \oplus \frac{1}{2}[\![Q]\!]\rangle$ are in $\mathscr{R}$, we have that

$$\langle \tfrac{1}{3}\delta(\tau\cdot(P \,{}_{\frac{1}{2}}\!\oplus Q)) \oplus \tfrac{2}{3}(\tfrac{1}{2}[\![P]\!] \oplus \tfrac{1}{2}[\![Q]\!]), \tfrac{1}{3}(\tfrac{1}{2}[\![P]\!] \oplus \tfrac{1}{2}[\![Q]\!]) \oplus \tfrac{2}{3}(\tfrac{1}{2}[\![P]\!] \oplus \tfrac{1}{2}[\![Q]\!]))\rangle \in cc(\mathscr{R}^\dagger).$$

Adding the pair of processes $\langle \delta(a\cdot(P \,{}_{\frac{1}{2}}\!\oplus Q)), \delta(a\cdot(\partial(\tau\cdot(P \,{}_{\frac{1}{2}}\!\oplus Q)) \,{}_{\frac{1}{3}}\!\oplus (P \,{}_{\frac{1}{2}}\!\oplus Q)))\rangle$ and closing for symmetry, then yields a branching probabilistic bisimulation relation relating $\delta(\mathbf{G_1})$ and $\delta(\mathbf{G_2})$.

(b) The $a$-derivatives of $\mathbf{I_1}$ and $\mathbf{I_2}$, i.e. the distributions $I_1' = \delta(b\cdot P+\tau\cdot Q)$ and $I_2' = \delta(\tau\cdot\partial(b\cdot P+\tau\cdot Q)+b\cdot P+\tau\cdot Q)$ are branching probabilistic bisimilar. A $\tau$-transition of $I_2'$ partially based on its left branch, can be simulated by $I_1'$ by a partial transition:

$$\begin{array}{lll} I_2' = r\cdot[\![I_2']\!]\oplus(1-r)\cdot[\![I_2']\!] & \xrightarrow{\tau} & r\cdot\delta(b\cdot P+\tau\cdot Q)\oplus(1-r)\cdot[\![Q]\!] \\ I_1' = r\cdot[\![I_1']\!]\oplus(1-r)\cdot[\![I_1']\!] & \xrightarrow{(\tau)} & r\cdot[\![I_1']\!]\oplus(1-r)\cdot[\![Q]\!] \;=\; r\cdot\delta(b\cdot P+\tau\cdot Q)\oplus(1-r)\cdot[\![Q]\!]. \end{array}$$

A $\tau$-transition of $I_1'$ can be directly simulated by $I_2'$ of course. It follows that the relation $\mathscr{R} = \{\langle\delta(\mathbf{I_1}),\delta(\mathbf{I_2})\rangle,\langle I_1',I_2'\rangle\}^\dagger \cup \{\,\langle\mu,\mu\rangle \mid \mu\in \mathit{Distr}(\mathscr{E})\,\}$, the symmetric relation containing the pairs mentioned and the diagonal of $\mathit{Distr}(\mathscr{E})$, constitutes a branching probabilistic bisimulation relation containing $\mathbf{I_1}$ and $\mathbf{I_2}$.

In the sequel we frequently need that probabilistic composition respects branching probabilistic bisimilarity of distributions, i.e. if, with respect to some index set $I$, we have distributions $\mu_i$ and $\nu_i$ such that $\mu_i \leftrightarrows_b \nu_i$ for $i\in I$, then also $\mu \leftrightarrows_b \nu$ for the distributions $\mu = \bigoplus_{i\in I} p_i\mu_i$ and $\nu = \bigoplus_{i\in I} p_i\nu_i$. The property directly follows from the following lemma, which is proven in [15].

**Lemma 10.** *Let distributions $\mu_1,\mu_2,\nu_1,\nu_2\in\mathit{Distr}(\mathscr{E})$ and $0\leqslant r\leqslant 1$ be such that $\mu_1\leftrightarrows_b\nu_1$ and $\mu_2\leftrightarrows_b\nu_2$. Then it holds that $\mu_1\,{}_r\!\oplus\,\mu_2\leftrightarrows_b\nu_1\,{}_r\!\oplus\,\nu_2$.*

We apply the above property in the proof of the next result. In the sequel any application of Lemma 10 will be done tacitly.

**Lemma 11.** *Let $\mu,\nu\in\mathit{Distr}(\mathscr{E})$ such that $\mu\leftrightarrows_b\nu$ and $\mu\Rightarrow\mu'$ for some $\mu'\in\mathit{Distr}(\mathscr{E})$. Then there are $\nu'\in\mathit{Distr}(\mathscr{E})$ such that $\nu\Rightarrow\nu'$ and $\mu'\leftrightarrows_b\nu'$.*

*Proof.* We check that a partial transition $\mu\xrightarrow{(\tau)}\mu'$ can be matched by $\nu$ given $\mu\leftrightarrows_b\nu$. So, suppose $\mu = \mu_1\,{}_r\!\oplus\,\mu_2$, $\mu_1\xrightarrow{\tau}\mu_1'$, and $\mu' = \mu_1'\,{}_r\!\oplus\,\mu_2$. By weak decomposability of $\leftrightarrows_b$ we can find distributions $\bar\nu,\nu_1,\nu_2$ such that $\nu\Rightarrow\bar\nu = \nu_1\,{}_r\!\oplus\,\nu_2$ and $\mu\leftrightarrows_b\bar\nu$, $\nu_1\leftrightarrows_b\mu_1$, $\nu_2\leftrightarrows_b\mu_2$. Choose distributions $\bar\nu_1,\bar\nu_1'$ such that $\nu_1\Rightarrow\bar\nu_1\xrightarrow{(\tau)}\nu_1'$ and $\bar\nu_1\leftrightarrows_b\mu_1$, $\nu_1'\leftrightarrows_b\mu_1'$. Put $\nu' = \nu_1'\,{}_r\!\oplus\,\nu_2$. Then $\nu\Rightarrow\nu'$, using Lemma 7c, and we have by Lemma 10 that $\nu' = \nu_1'\,{}_r\!\oplus\,\nu_2\leftrightarrows_b\mu_1'\,{}_r\!\oplus\,\mu_2 = \mu'$ since $\nu_1'\leftrightarrows_b\mu_1'$ and $\nu_2\leftrightarrows_b\mu_2$. □

## 5 Branching probabilistic bisimilarity is continuous

Fix a finite set of non-deterministic processes $\mathscr{F}\subseteq\mathscr{E}$ that is *transition closed*, in the sense that if $E\in\mathscr{F}$ and $E\xrightarrow{\alpha}\bigoplus_{i\in I}p_i\cdot F_i$ then also $F_i\in\mathscr{F}$. Consequently, if $\mu\in\mathit{Distr}(\mathscr{F})$ and $\mu\xrightarrow{(\alpha)}\mu'$ then $\mu'\in\mathit{Distr}(\mathscr{F})$. Also, if $\mu\in\mathit{Distr}(\mathscr{F})$ and $\mu\Rightarrow\bar\mu$ then $\bar\mu\in\mathit{Distr}(\mathscr{F})$. By Theorem 1 $\mathit{Distr}(\mathscr{F})$ is a sequentially compact subspace of the complete metric space $\mathit{Distr}(\mathscr{E})$, meaning that every sequence $(\mu_i)_{i=0}^\infty$ in $\mathit{Distr}(\mathscr{F})$ has a subsequence $(\mu_{i_k})_{k=0}^\infty$ such that $\lim_{k\to\infty}\mu_{i_k}=\mu$ for some distribution $\mu\in\mathit{Distr}(\mathscr{F})$. In particular, if $\lim_{i\to\infty}\mu_i=\mu$ and $\mu_i\in\mathit{Distr}(\mathscr{F})$, then also $\mu\in\mathit{Distr}(\mathscr{F})$, i.e. $\mathit{Distr}(\mathscr{F})$ is a closed subset of $\mathit{Distr}(\mathscr{E})$. Due to the finitary nature of our process algebra, each distribution $\mu\in\mathit{Distr}(\mathscr{E})$ occurs in $\mathit{Distr}(\mathscr{F})$ for some such $\mathscr{F}$, based on $\mathit{spt}(\mu)$.

In the following three lemmas we establish a number of continuity results. Assume $\lim_{i\to\infty}\nu_i=\nu$. Then Lemma 12 states that, for a Dirac distribution $\delta(E)$, if $\delta(E)\xrightarrow{\alpha}\nu_i$ for $i\in\mathbb{N}$ then also $\delta(E)\xrightarrow{\alpha}\nu$. Lemma 13 extends this and shows that, for a general distribution $\mu$, if $\mu\xrightarrow{\alpha}\nu_i$ for $i\in\mathbb{N}$ then $\mu\xrightarrow{\alpha}\nu$. Finally, Lemma 14 establishes the limit case: if $\lim_{i\to\infty}\mu_i=\mu$ and $\mu_i\xrightarrow{\alpha}\nu_i$ for $i\in\mathbb{N}$ then $\mu\xrightarrow{\alpha}\nu$.

**Lemma 12.** *Let $E\in\mathscr{F}$ be a non-deterministic process, $\alpha\in\mathscr{A}$ an action, $(\nu_i)_{i=0}^\infty\in\mathit{Distr}(\mathscr{F})^\infty$ an infinite sequence in $\mathit{Distr}(\mathscr{F})$, and $\nu\in\mathit{Distr}(\mathscr{F})$ a distribution satisfying $\lim_{i\to\infty}\nu_i=\nu$. If, for all $i\in\mathbb{N}$, $\delta(E)\xrightarrow{(\alpha)}\nu_i$ then it holds that $\delta(E)\xrightarrow{(\alpha)}\nu$.*

*Proof.* For $E \in \mathscr{F}$ and $\alpha \in \mathscr{A}$, define $E\upharpoonright\alpha = cc(\{\, \mu \mid E \xrightarrow{\alpha} \mu\})$, pronounced $E$ 'after' $\alpha$, to be the convex closure in $Distr(\mathscr{E})$ of all distributions that can be reached from $E$ by an $\alpha$-transition. Then $\delta(E) \xrightarrow{\alpha} \nu$ iff $\nu \in E\upharpoonright\alpha$. Recall that transitions for non-deterministic processes are not probabilistically combined. See Definition 5. Since $E\upharpoonright\alpha \subseteq Distr(\mathscr{F})$ is the convex closure of a finite set of distributions, it is certainly closed in the space $Distr(\mathscr{F})$. Since it holds that $\delta(E) \xrightarrow{\alpha} \nu_i$ for all $i \in \mathbb{N}$, one has $\nu_i \in E\upharpoonright\alpha$ for $i \in \mathbb{N}$. Hence, $\lim_{i\to\infty} \nu_i = \nu$ implies that $\nu \in E\upharpoonright\alpha$, i.e. $\delta(E) \xrightarrow{\alpha} \nu$.

For $E \in \mathscr{F}$, define $E\upharpoonright(\tau) := cc(\{\mu \mid E \xrightarrow{\tau} \mu\} \cup \{E\})$. Then $\delta(E) \xrightarrow{(\tau)} \nu$ iff $\nu \in E\upharpoonright(\tau)$. The set $E\upharpoonright(\tau) \subseteq Distr(\mathscr{F})$ is closed, and thus $\nu_i \in E\upharpoonright(\tau)$ implies $\nu \in E\upharpoonright(\tau)$, which means $\delta(E) \xrightarrow{(\tau)} \nu$. □

The above result for Dirac distributions holds for general distributions as well.

**Lemma 13.** *Let $\mu, \nu \in Distr(\mathscr{F})$, $\alpha \in \mathscr{A}$, $(\nu_i)_{i=0}^{\infty} \in Distr(\mathscr{F})^{\infty}$, and assume $\lim_{i\to\infty} \nu_i = \nu$. If it holds that $\mu \xrightarrow{(\alpha)} \nu_i$ for all $i \in \mathbb{N}$, then also $\mu \xrightarrow{(\alpha)} \nu$.*

*Proof.* Suppose $\mu \xrightarrow{(\alpha)} \nu_i$ for all $i \in I$. Let $\mu = \bigoplus_{j=1}^{k} p_j \cdot E_j$. By Lemma 8, for all $i \in \mathbb{N}$ and $1 \leqslant j \leqslant k$ there are $\nu_{ij}$ such that $\delta(E_j) \xrightarrow{(\alpha)} \nu_{ij}$ and $\nu_i = \bigoplus_{j=1}^{k} p_j \cdot \nu_{ij}$. The countable sequence $(\nu_{i1}, \nu_{i2}, \ldots, \nu_{ik})_{i=0}^{\infty}$ of $k$-dimensional vectors of probability distributions need not have a limit. However, by the sequential compactness of $Distr(\mathscr{F})$ this sequence has an infinite subsequence in which the first components $\nu_{i_1}$ converge to a limit $\eta_1$. That sequence in turn has an infinite subsequence in which also the second components $\nu_{i_2}$ converge to a limit $\eta_2$. Going on this way, one finds a subsequence $(\nu_{i_h 1}, \nu_{i_h 2}, \ldots, \nu_{i_h k})_{h=0}^{\infty}$ of $(\nu_{i1}, \nu_{i2}, \ldots, \nu_{ik})_{i=0}^{\infty}$ for $i_0 < i_1 < \ldots$ that has a limit, say $\lim_{h\to\infty}(\nu_{i_h 1}, \nu_{i_h 2}, \ldots, \nu_{i_h k}) = (\eta_1, \eta_2, \ldots, \eta_k)$. Using that $\lim_{h\to\infty} \nu_{i_h} = \nu$, one obtains $\nu = \bigoplus_{j=1}^{k} p_j \cdot \eta_j$. For each $j = 1, \ldots, k$, by Lemma 12, since $\delta(E_j) \xrightarrow{(\alpha)} \nu_{ij}$ for all $i \in I$ and $\lim_{h\to\infty} \nu_{i_h j} = \eta_j$, we conclude that $\delta(E_j) \xrightarrow{(\alpha)} \eta_j$. Thus, by Lemma 7, $\mu = \bigoplus_{j=1}^{k} p_j \cdot E_j \xrightarrow{(\alpha)} \bigoplus_{j=1}^{k} p_j \cdot \eta_j = \nu$. □

Next, we consider a partial transition over a convergent sequence of distributions.

**Lemma 14.** *Let $(\mu_i)_{i=0}^{\infty}, (\nu_i)_{i=0}^{\infty} \in Distr(\mathscr{F})^{\infty}$ such that $\lim_{i\to\infty} \mu_i = \mu$ and $\lim_{i\to\infty} \nu_i = \nu$. If it holds that $\mu_i \xrightarrow{(\alpha)} \nu_i$ for all $i \in \mathbb{N}$, then also $\mu \xrightarrow{(\alpha)} \nu$.*

*Proof.* Since $\lim_{i\to\infty} \mu_i = \mu$, we can write $\mu_i = (1 - r_i)\mu \oplus r_i \mu_i''$, for suitable $\mu_i'' \in Distr(\mathscr{F})$ and $r_i \geqslant 0$ such that $\lim_{i\to\infty} r_i = 0$, as guaranteed by Lemma 2. Because $\mu_i \xrightarrow{(\alpha)} \nu_i$, by Lemma 8 there are distributions $\nu_i', \nu_i'' \in Distr(\mathscr{F})$ for $i \in \mathbb{N}$ such that $\nu_i = (1 - r_i)\nu_i' \oplus r_i \nu_i''$, $\mu \xrightarrow{(\alpha)} \nu_i'$, and $\mu_i'' \xrightarrow{(\alpha)} \nu_i''$. We have $\lim_{i\to\infty} \nu_i' = \nu$ as well, since $\lim_{i\to\infty} r_i = 0$. Thus, $\lim_{i\to\infty} \nu_i' = \nu$ and $\mu \xrightarrow{(\alpha)} \nu_i'$ for $i \in \mathbb{N}$. Therefore, it follows by Lemma 13 that $\mu \xrightarrow{(\alpha)} \nu$. □

For $\mu, \nu \in Distr(\mathscr{F})$, we write $\mu \Rightarrow_n \nu$ if there are $\eta_0, \eta_1, \ldots, \eta_n \in Distr(\mathscr{F})$ such that $\mu = \eta_0 \xrightarrow{(\tau)} \eta_1 \xrightarrow{(\tau)} \ldots \xrightarrow{(\tau)} \eta_n = \nu$. Clearly, it holds that $\mu \Rightarrow_n \nu$ for some $n \in \mathbb{N}$ in case $\mu \Rightarrow \nu$, because $\Rightarrow$ is the transitive closure of $\xrightarrow{(\tau)}$.

We have the following pendant of Lemma 14 for $\Rightarrow_n$.

**Lemma 15.** *Let $(\mu_i)_{i=0}^{\infty}, (\nu_i)_{i=0}^{\infty} \in Distr(\mathscr{F})^{\infty}$, $\lim_{i\to\infty} \mu_i = \mu$ and $\lim_{i\to\infty} \nu_i = \nu$. If $\mu_i \Rightarrow_n \nu_i$ for all $i \in \mathbb{N}$ then $\mu \Rightarrow_n \nu$.*

*Proof.* By induction on $n$. Basis, $n = 0$: Trivial. Induction step, $n+1$: Given $(\mu_i)_{i=0}^{\infty}, (\nu_i)_{i=0}^{\infty} \in Distr(\mathscr{F})^{\infty}$, $\mu = \lim_{i\to\infty} \mu_i$, and $\nu = \lim_{i\to\infty} \nu_i$, suppose $\mu_i \Rightarrow_{n+1} \nu_i$ for all $i \in \mathbb{N}$. Let $(\eta_i)_{i=0}^{\infty} \in Distr(\mathscr{F})^{\infty}$ be such that $\mu_i \xrightarrow{(\tau)} \eta_i \Rightarrow_n \nu_i$ for all $i \in \mathbb{N}$. Since $Distr(\mathscr{F})$ is sequentially compact, the sequence $(\eta_i)_{i=0}^{\infty}$ has a convergent subsequence $(\eta_{i_k})_{k=0}^{\infty}$; put $\eta = \lim_{k\to\infty} \eta_{i_k}$. Because $\mu_{i_k} \xrightarrow{(\tau)} \eta_{i_k}$ for all $k \in \mathbb{N}$, one has $\mu \xrightarrow{(\tau)} \eta$ by Lemma 14. Since $\eta_{i_k} \Rightarrow_n \nu_{i_k}$ for $k \in \mathbb{N}$, the induction hypothesis yields $\eta \Rightarrow_n \nu$. It follows that $\mu \Rightarrow_{n+1} \nu$. □

We adapt Lemma 15 to obtain a continuity result for weak transitions $\Rightarrow$.

**Lemma 16.** *Let $(\mu_i)_{i=0}^{\infty}, (\nu_i)_{i=0}^{\infty} \in Distr(\mathscr{F})^{\infty}$, $\lim_{i\to\infty} \mu_i = \mu$ and $\lim_{i\to\infty} \nu_i = \nu$. If $\mu_i \Rightarrow \nu_i$ for all $i \in \mathbb{N}$, then $\mu \Rightarrow \nu$.*

*Proof.* Since $\mathscr{F}$ contains only finitely many non-deterministic processes, which can do finitely many $\tau$-transitions only, a global upperbound $N$ exists such that if $\mu \Rightarrow \nu$ then $\mu \Rightarrow_k \nu$ for some $k \leqslant N$.

Moreover, as each sequence $\mu = \eta_0 \xrightarrow{(\tau)} \eta_1 \xrightarrow{(\tau)} \ldots \xrightarrow{(\tau)} \eta_k = \nu$ with $k < N$ can be extended to a sequence $\mu = \eta_0 \xrightarrow{(\tau)} \eta_1 \xrightarrow{(\tau)} \ldots \xrightarrow{(\tau)} \eta_N = \nu$, namely by taking $\eta_i = \nu$ for all $k < i \leqslant N$, on $\mathscr{F}$ the relations $\Rightarrow$ and $\Rightarrow_N$ coincide. Consequently, Lemma 16 follows from Lemma 15. □

The following theorem says that equivalence classes of branching probabilistic bisimilarity in $Distr(\mathscr{F})$ are closed sets of distributions.

**Theorem 17.** *Let $\hat{\mu}, \hat{\nu} \in Distr(\mathscr{F})$ and $(\nu_i)_{i=0}^{\infty} \in Distr(\mathscr{F})^{\infty}$ such that $\hat{\mu} \leftrightarrows_b \nu_i$ for all $i \in \mathbb{N}$ and $\hat{\nu} = \lim_{i\to\infty} \nu_i$. Then it holds that $\hat{\mu} \leftrightarrows_b \hat{\nu}$.*

*Proof.* Define the relation $\mathscr{R}$ on $Distr(\mathscr{F})$ by

$$\begin{aligned} \mu \mathscr{R} \nu \iff\ & \exists (\mu_i)_{i=0}^{\infty}, (\nu_i)_{i=0}^{\infty} \in Distr(\mathscr{F})^{\infty}: \\ & \lim_{i\to\infty} \mu_i = \mu \wedge \lim_{i\to\infty} \nu_i = \nu \wedge \forall i \in \mathbb{N}\colon\ \mu_i \leftrightarrows_b \nu_i \end{aligned}$$

As $\hat{\mu} \mathscr{R} \hat{\nu}$ (taking $\mu_i := \hat{\mu}$ for all $i \in I$), it suffices to show that $\mathscr{R}$ is a branching probabilistic bisimulation.

Suppose $\mu \mathscr{R} \nu$. Let $(\mu_i)_{i=0}^{\infty}, (\nu_i)_{i=0}^{\infty} \in Distr(\mathscr{F})^{\infty}$ be such that $\lim_{i\to\infty} \mu_i = \mu$, $\lim_{i\to\infty} \nu_i = \nu$, and $\mu_i \leftrightarrows_b \nu_i$ for all $i \in \mathbb{N}$. Since $\lim_{i\to\infty} \mu_i = \mu$, there exist $(\mu_i')_{i=0}^{\infty} \in Distr(\mathscr{F})^{\infty}$ and $(r_i)_{i=0}^{\infty} \in \mathbb{R}^{\infty}$ such that $\mu_i = (1-r_i)\mu \oplus r_i \mu_i'$ for all $i \in \mathbb{N}$ and $\lim_{i\to\infty} r_i = 0$.

(i) Towards weak decomposability of $\mathscr{R}$ for $\mu$ vs. $\nu$, suppose $\mu = \bigoplus_{j\in J} q_j \cdot \bar{\mu}_j$. So, for all $i \in \mathbb{N}$, we have that $\mu_i = (1-r_i)\big(\bigoplus_{j\in J} q_j \cdot \bar{\mu}_j\big) \oplus r_i \mu_i'$. By weak decomposability of $\leftrightarrows_b$, there exist $\bar{\bar{\nu}}_i$, $\nu_i'$ and $\nu_{ij}$ for $i \in \mathbb{N}$ and $j \in J$ such that $\nu_i \Rightarrow \bar{\bar{\nu}}_i$, $\mu_i \leftrightarrows_b \bar{\bar{\nu}}_i$, $\bar{\bar{\nu}}_i = (1-r_i)\big(\bigoplus_{j\in J} q_j \cdot \nu_{ij}\big) \oplus r_i \nu_i'$, $\mu_i' \leftrightarrows_b \nu_i'$, and $\bar{\mu}_j \leftrightarrows_b \nu_{ij}$ for $j \in J$.

The sequences $(\nu_{ij})_{i=0}^{\infty}$ for $j \in J$ may not converge. However, by sequential compactness of $Distr(\mathscr{F})$ (and successive sifting out for each $j \in J$) an index sequence $(i_k)_{k=0}^{\infty}$ exists such that the sequences $(\nu_{i_k j})_{k=0}^{\infty}$ converge, say $\lim_{k\to\infty} \nu_{i_k j} = \bar{\nu}_j$ for $j \in J$. Put $\bar{\nu} = \bigoplus_{j\in J} q_j \cdot \bar{\nu}_j$. Then it holds that

$$\lim_{k\to\infty} \bar{\bar{\nu}}_{i_k} = \lim_{k\to\infty} (1-r_{i_k})\big(\bigoplus_{j\in J} q_j \cdot \nu_{i_k j}\big) \oplus r_{i_k} \nu_{i_k}' = \lim_{k\to\infty} \bigoplus_{j\in J} q_j \cdot \nu_{i_k j} = \bigoplus_{j\in J} q_j \cdot \bar{\nu}_j = \bar{\nu}$$

as $\lim_{k\to\infty} r_{i_k} = 0$ and probabilistic composition is continuous. Since $\nu_{i_k} \Rightarrow \bar{\bar{\nu}}_{i_k}$ for all $k \in \mathbb{N}$, one has $\lim_{k\to\infty} \nu_{i_k} \Rightarrow \lim_{k\to\infty} \bar{\bar{\nu}}_{i_k}$, i.e. $\nu \Rightarrow \bar{\nu}$, by Lemma 16. Also, $\mu_{i_k} \leftrightarrows_b \bar{\bar{\nu}}_{i_k}$ for all $k \in \mathbb{N}$. Therefore, by definition of $\mathscr{R}$, we obtain $\mu \mathscr{R} \bar{\nu}$. Since $\bar{\mu}_j \leftrightarrows_b \nu_{i_k j}$ for all $k \in \mathbb{N}$ and $j \in J$, it follows that $\bar{\mu}_j \mathscr{R} \bar{\nu}_j$ for $j \in J$. Thus, $\nu \Rightarrow \bar{\nu} = \bigoplus_{j\in J} q_j \cdot \bar{\nu}_j$, $\mu \mathscr{R} \bar{\nu}$, and $\bar{\mu}_j \mathscr{R} \bar{\nu}_j$ for all $j \in J$, as was to be shown. Hence the relation $\mathscr{R}$ is weakly decomposable.

(ii) For the transfer property, suppose $\mu \xrightarrow{\alpha} \mu'$ for some $\alpha \in \mathscr{A}$. Since, for each $i \in \mathbb{N}$, $\mu_i \leftrightarrows_b \nu_i$ and $\mu_i = (1-r_i)\mu \oplus r_i \mu_i'$, it follows from weak decomposability of $\leftrightarrows_b$ that distributions $\bar{\nu}_i$, $\nu_i'$ and $\nu_i''$ exist such that $\nu_i \Rightarrow \bar{\nu}_i$, $\mu_i \leftrightarrows_b \bar{\nu}_i$, $\bar{\nu}_i = (1-r_i)\nu_i' \oplus r_i \nu_i''$ and $\mu \leftrightarrows_b \nu_i'$. By the transfer property for $\leftrightarrows_b$, for each $i \in \mathbb{N}$ exist $\bar{\eta}_i, \eta_i' \in Distr(\mathscr{E})$ such that

$$\nu_i' \Rightarrow \bar{\eta}_i,\ \bar{\eta}_i \xrightarrow{(\alpha)} \eta_i',\ \mu \leftrightarrows_b \bar{\eta}_i, \text{ and } \mu' \leftrightarrows_b \eta_i'.$$

We have $\bar{\nu}_i' \in Distr(\mathscr{F})$ for $i \in \mathbb{N}$. Also, $\bar{\eta}_i, \eta_i' \in Distr(\mathscr{F})$ for $i \in \mathbb{N}$, since $\mathscr{F}$ is assumed to be transition closed. Therefore, by sequential compactness of $Distr(\mathscr{F})$, the sequences $(\bar{\nu}_i')_{i=0}^{\infty}$, $(\bar{\eta}_i)_{i=0}^{\infty}$,

$(\bar\eta'_i)_{i=0}^{\infty}$ have converging subsequences $(\bar\nu'_{i_k})_{k=0}^{\infty}$, $(\bar\eta_{i_k})_{k=0}^{\infty}$, and $(\bar\eta'_{i_k})_{k=0}^{\infty}$, respectively. Put $\bar\nu = \lim_{k\to\infty}\nu'_{i_k}$, $\bar\eta = \lim_{k\to\infty}\bar\eta_{i_k}$, and $\eta' = \lim_{k\to\infty}\eta'_{i_k}$. As $\lim_{k\to\infty} r_{i_k} = 0$, one has $\lim_{k\to\infty}\bar\nu_{i_k} = \lim_{k\to\infty}\nu'_{i_k} = \bar\nu$.

Since $\nu_{i_k} \Rightarrow \bar\nu_{i_k}$ for $k\in\mathbb{N}$, we obtain $\lim_{k\to\infty}\nu_{i_k} \Rightarrow \lim_{k\to\infty}\bar\nu_{i_k}$ by Lemma 16, thus $\nu\Rightarrow\bar\nu$. Likewise, as $\nu'_{i_k} \Rightarrow \bar\eta_{i_k}$ for all $k\in\mathbb{N}$, one has $\bar\nu\Rightarrow\bar\eta$, and therefore $\nu\Rightarrow\bar\eta$. Furthermore, because $\bar\eta_{i_k} \xrightarrow{(\alpha)} \eta'_{i_k}$ for $k\in\mathbb{N}$, it follows that $\bar\eta \xrightarrow{(\alpha)} \eta'$, now by Lemma 14. From $\mu \leftrightarrows_b \bar\eta_{i_k}$ for all $k\in\mathbb{N}$, we obtain $\mu\,\mathscr{R}\,\bar\eta$ by definition of $\mathscr{R}$. Finally, $\mu' \leftrightarrows_b \eta'_{i_k}$ for all $k\in\mathbb{N}$ yields $\mu'\,\mathscr{R}\,\eta'$. Thus $\nu\Rightarrow\bar\eta \xrightarrow{(\alpha)} \eta'$, $\mu\,\mathscr{R}\,\bar\eta$, and $\mu'\,\mathscr{R}\,\bar\eta'$, which was to be shown. □

The following corollary of Theorem 17 will be used in the next section.

**Corollary 18.** *For each $\mu\in Distr(\mathscr{E})$, the set $T_\mu = \{\,\nu\in Distr(\mathscr{E}) \mid \nu \leftrightarrows_b \mu\wedge\mu\Rightarrow\nu\,\}$ is a sequentially compact set.*

*Proof.* For $\mu = \bigoplus_{i\in I} p_i\cdot E_i$, the set of processes $\mathscr{F} = \{\,E\in\mathscr{E} \mid E \text{ occurs in } E_i \text{ for some } i\in I\,\}$ is finite and closed under transitions. Clearly, $\mu\in Distr(\mathscr{F})$. Moreover, $Distr(\mathscr{F})$ is a sequentially compact subset of $Distr(\mathscr{E})$. Taking $\mu_i = \mu$ for all $i\in\mathbb{N}$ in Lemma 16 yields that $\{\,\nu \mid \mu\Rightarrow\nu\,\}$ is a closed subset of $Distr(\mathscr{F})$. Similarly, the set $\{\,\nu \mid \nu\leftrightarrows_b\mu\,\}$ is a closed subset of $Distr(\mathscr{F})$ by Theorem 17. The statement then follows since the intersection of two closed subsets of $Distr(\mathscr{F})$ is itself closed, and hence sequentially compact. □

# 6 Cancellativity for branching probabilistic bisimilarity

With the results of Section 5 in place, we turn to stable processes and cancellativity. In the introduction we argued that in general it doesn't need to be the case that two branching probabilistic bisimilar distributions assign the same weight to equivalence classes. Here we show that this property does hold when restricting to stable distributions. We continue to prove the announced unfolding result, that for every distribution $\mu$ there exists a stable distribution $\sigma$ such that $\mu\Rightarrow\sigma$ and $\mu\leftrightarrows_b\sigma$. That result will be pivotal in the proof of the cancellation theorem, Theorem 22.

**Definition 19.** *A distribution $\mu\in Distr(\mathscr{E})$ is called* stable *if, for all $\bar\mu\in Distr(\mathscr{E})$, $\mu\Rightarrow\bar\mu$ and $\mu\leftrightarrows_b\bar\mu$ imply that $\bar\mu=\mu$.*

Thus, a distribution $\mu$ is called stable if it cannot perform internal activity without leaving its branching bisimulation equivalence class. By definition of $\xrightarrow{(\tau)}$ it is immediate that if $\bigoplus_{i\in I} p_i\cdot\mu_i$ is a stable distribution with $p_i>0$ for $i\in I$, then also each probabilistic component $\mu_i$ is stable. Also, because two stable distributions $\mu$ and $\nu$ don't have any non-trivial partial $\tau$-transitions, weak decomposability between them amounts to decomposability, i.e. if $\mu\leftrightarrows_b\nu$ and $\mu=\bigoplus_{i\in I}p_i\mu_i$ then distributions $\nu_i$ for $i\in I$ exist such that $\nu=\bigoplus_{i\in I}p_i\nu_i$ and $\mu_i\leftrightarrows_b\nu_i$ for $i\in I$.

The next result states that, contrary to distributions in general, two stable distributions are branching bisimilar precisely when they assign the same probability on all branching bisimilarity classes of $\mathscr{E}$.

**Lemma 20.** *Let $\mu,\nu\in Distr(\mathscr{E})$ be two stable distributions. Then it holds that $\mu\leftrightarrows_b\nu$ iff $\mu[C]=\nu[C]$ for each equivalence class $C$ of branching probabilistic bisimilarity in $\mathscr{E}$.*

*Proof.* Suppose $\mu=\bigoplus_{i\in I}p_i\cdot E_i$, $\nu=\bigoplus_{j\in J}q_j\cdot F_j$, and $\mu\leftrightarrows_b\nu$. By weak decomposability, $\nu\Rightarrow\bar\nu = \bigoplus_{i\in I}p_i\cdot\nu_i$ for suitable $\nu_i\in Distr(\mathscr{E})$ for $i\in I$ with $\nu_i\leftrightarrows_b\delta(E_i)$ and $\bar\nu\leftrightarrows_b\mu$. Hence, $\bar\nu\leftrightarrows_b\mu\leftrightarrows_b\nu$. Thus, by stability of $\nu$, we have $\bar\nu=\nu$. Say, $\nu_i=\bigoplus_{j\in J}q_{ij}\cdot F_j$ with $q_{ij}\geqslant 0$, for $i\in I$, $j\in J$. Since $\nu_i\leftrightarrows_b\delta(E_i)$, we have by weak decomposability, $\delta(E_i)\Rightarrow\bigoplus_{j\in J}q_{ij}\cdot\mu'_{ij}$ such that $\delta(E_i)\leftrightarrows_b\bigoplus_{j\in J}q_{ij}\cdot\mu'_{ij}$

and $\mu'_{ij} \leftrightarrows_b \delta(F_j)$ for suitable $\mu'_{ij} \in \mathit{Distr}(\mathscr{E})$. Since $\mu$ is stable, so is $\delta(E_i)$. Hence $\delta(E_i) = \bigoplus_{j\in J} q_{ij}\cdot\mu'_{ij}$, $\mu'_{ij} = \delta(E_i)$, and $E_i \leftrightarrows_b F_j$ if $q_{ij} > 0$. Put $p_{ij} = p_i q_{ij}$, $E_{ij} = E_i$ if $q_{ij} > 0$, and $E_{ij} = \mathbf{0}$ otherwise, $F_{ij} = F_j$ if $q_{ij} > 0$, and $F_{ij} = \mathbf{0}$ otherwise, for $i \in I$, $j \in J$. Then it holds that

$$\mu = \textstyle\bigoplus_{i\in I} p_i\cdot E_i = \bigoplus_{i\in I} p_i\cdot\big(\bigoplus_{j\in J} q_{ij}\cdot E_i\big) = \bigoplus_{i\in I}\bigoplus_{j\in J} p_i q_{ij}\cdot E_i = \bigoplus_{i\in I}\bigoplus_{j\in J} p_{ij}\cdot E_{ij}$$
$$\nu = \textstyle\bigoplus_{i\in I} p_i\cdot\nu_i \;= \bigoplus_{i\in I} p_i\cdot\big(\bigoplus_{j\in J} q_{ij}\cdot F_j\big) = \bigoplus_{i\in I}\bigoplus_{j\in J} p_i q_{ij}\cdot F_j = \bigoplus_{i\in I}\bigoplus_{j\in J} p_{ij}\cdot F_{ij}.$$

Now, for any equivalence class $C$ of $\mathscr{E}$ modulo $\leftrightarrows_b$, it holds that $E_{ij} \in C \Leftrightarrow F_{ij} \in C$ for all indices $i \in I$, $j \in J$. So, $\mu[C] = \sum_{i\in I, j\in J:\ E_{ij}\in C} p_{ij} = \sum_{i\in I, j\in J:\ F_{ij}\in C} p_{ij} = \nu[C]$.

For the reverse direction, suppose $\mu = \bigoplus_{i\in I} p_i\cdot E_i$, $\nu = \bigoplus_{j\in J} q_j\cdot F_j$, with $p_i, q_j > 0$, and $\mu[C] = \nu[C]$ for each equivalence class $C \in \mathscr{E}/\leftrightarrows$.

For $i \in I$ and $j \in J$, let $C_i$ and $D_j$ be the equivalence class in $\mathscr{E}$ of $E_i$ and $F_j$ modulo $\leftrightarrows_b$. Define $r_{ij} = \delta_{ij} p_i q_j / \mu[C_i]$, for $i \in I$, $j \in J$, where $\delta_{ij} = 1$ if $E_i \leftrightarrows_b F_j$ and $\delta_{ij} = 0$ otherwise. Then it holds that

$$\textstyle\sum_{j\in J} r_{ij} = \sum_{j\in J} \frac{\delta_{ij} p_i q_j}{\mu[C_i]} = \frac{p_i}{\mu[C_i]} \sum_{j\in J} \delta_{ij} q_j = \frac{p_i \nu[C_i]}{\mu[C_i]} = p_i.$$

Since $\delta_{ij} p_i q_j / \mu[C_i] = \delta_{ij} p_i q_j / \nu[D_j]$ for $i \in I$, $j \in J$, we also have $\sum_{i\in I} r_{ij} = q_j$. Therefore, we can write $\mu = \bigoplus_{i\in I}\bigoplus_{j\in J} r_{ij}\cdot E_{ij}$ and $\nu = \bigoplus_{i\in I}\bigoplus_{j\in J} r_{ij}\cdot F_{ij}$ for suitable $E_{ij}$ and $F_{ij}$ such that $E_{ij} \leftrightarrows_b F_{ij}$. Calling Lemma 10 it follows that $\mu \leftrightarrows_b \nu$. □

Next, in Lemma 21, we are about to prove a crucial property for our proof of cancellativity, the proof of Theorem 22 below. Generally, a distribution may allow inert partial transitions. However, the distribution can be unfolded to reach via inert partial transitions a stable distribution, which doesn't have these by definition. To obtain the result we will rely on the topological property of sequential compactness of the set $T_\mu = \{\, \mu' \mid \mu' \leftrightarrows_b \mu \wedge \mu \Rightarrow \mu' \,\}$ introduced in the previous section.

**Lemma 21.** *For all $\mu \in \mathit{Distr}(\mathscr{E})$ there is a stable distribution $\sigma \in \mathit{Distr}(\mathscr{E})$ such that $\mu \Rightarrow \sigma$.*

*Proof.* Define the *weight* of a distribution by $wgt(\mu) = \sum_{E\in\mathscr{E}} \mu(E)\cdot c(E)$, i.e., the weighted average of the complexities of the states in its support. In view of these definitions, $E \xrightarrow{\alpha} \mu$ implies $wgt(\mu) < wgt(\delta(E))$ and $\mu \xrightarrow{\alpha} \mu'$ implies $wgt(\mu') < wgt(\mu)$. In addition, $\mu \Rightarrow \mu'$ implies $wgt(\mu') \leqslant wgt(\mu)$.

For a distribution $\mu \in \mathit{Distr}(\mathscr{E})$, the set $T_\mu$ is given by $T_\mu = \{\, \mu' \mid \mu' \leftrightarrows_b \mu \wedge \mu \Rightarrow \mu' \,\}$. Consider the value $\inf\{\, wgt(\mu') \mid \mu' \in T_\mu \,\}$. By Corollary 18, $T_\mu$ is a sequentially compact set. Since the infimum over a sequentially compact set will be reached, there exists a distribution $\sigma$ such that $\mu \Rightarrow \sigma$, $\sigma \leftrightarrows_b \mu$, and $wgt(\sigma) = \inf\{\, wgt(\mu') \mid \mu' \in T_\mu \,\}$. By definition of $T_\mu$, the distribution $\sigma$ must be stable. □

We have arrived at the main result of the paper, slightly more general formulated compared to the description in the introduction. The message remains the same: if two distributions are branching probabilistic bisimilar and have components that are branching probabilistic bisimilar, then the components that remain after cancelling the earlier components are also branching probabilistic bisimilar. As we see, the previous lemma is essential in the proof as given.

**Theorem 22** (Cancellativity)**.** *Let $\mu, \mu', \nu, \nu' \in \mathit{Distr}(\mathscr{E})$ and $0 < r \leqslant 1$ be such that $\mu \,{}_r\!\oplus\, \nu \leftrightarrows_b \mu' \,{}_r\!\oplus\, \nu'$ and $\nu \leftrightarrows_b \nu'$. Then it holds that $\mu \leftrightarrows_b \mu'$.*

*Proof.* Choose $\mu$, $\mu'$, $\nu$, $\nu'$, and $r$ according to the premise of the theorem. By Lemma 21, a stable distribution $\sigma$ exists such that $\mu \,{}_r\!\oplus\, \nu \Rightarrow \sigma$ and $\sigma \leftrightarrows_b \mu \,{}_r\!\oplus\, \nu$. By weak decomposability, we can find distributions $\bar\mu$ and $\bar\nu$ such that $\sigma \Rightarrow \bar\mu \,{}_r\!\oplus\, \bar\nu$, $\bar\mu \leftrightarrows_b \mu$, and $\bar\nu \leftrightarrows_b \nu$. By stability of $\sigma$ we have $\sigma = \bar\mu \,{}_r\!\oplus\, \bar\nu$.

Thus $\bar{\mu}\,{}_r\!\oplus\,\bar{\nu}$ is stable. Symmetrically, there are distributions $\bar{\mu}'$ and $\bar{\nu}'$ such that $\bar{\mu}' \leftrightarrows_b \mu'$, $\bar{\nu}' \leftrightarrows_b \nu'$ and such that $\bar{\mu}'\,{}_r\!\oplus\,\bar{\nu}'$ is stable. Note, $\bar{\mu}\,{}_r\!\oplus\,\bar{\nu} \leftrightarrows_b \mu\,{}_r\!\oplus\,\nu \leftrightarrows_b \mu'\,{}_r\!\oplus\,\nu' \leftrightarrows_b \bar{\mu}'\,{}_r\!\oplus\,\bar{\nu}'$.

Let $C \subseteq \mathscr{E}$ be an equivalence class of $\mathscr{E}/\leftrightarrows_b$. The distributions $\bar{\mu}\,{}_r\!\oplus\,\bar{\nu}$ and $\bar{\mu}'\,{}_r\!\oplus\,\bar{\nu}'$ are stable and $\bar{\mu}\,{}_r\!\oplus\,\bar{\nu} \leftrightarrows_b \bar{\mu}'\,{}_r\!\oplus\,\bar{\nu}'$. From Lemma 20 we obtain that $(\bar{\mu}\,{}_r\!\oplus\,\bar{\nu})[C] = (\bar{\mu}'\,{}_r\!\oplus\,\bar{\nu}')[C]$. Since $\nu$ and $\bar{\nu}$ are stable and $\bar{\nu} \leftrightarrows_b \bar{\nu}'$, we have $\bar{\nu}[C] = \bar{\nu}'[C]$ for the same reason. Because $(\bar{\mu}\,{}_r\!\oplus\,\bar{\nu})[C] = r\cdot\bar{\mu}[C] + (1-r)\cdot\bar{\nu}[C]$ and $(\bar{\mu}'\,{}_r\!\oplus\,\bar{\nu}')[C] = r\cdot\bar{\mu}'[C] + (1-r)\cdot\bar{\nu}'[C]$, we calculate

$$r\cdot\bar{\mu}[C] = (\bar{\mu}\,{}_r\!\oplus\,\bar{\nu})[C] - (1-r)\cdot\bar{\nu}[C] = (\bar{\mu}'\,{}_r\!\oplus\,\bar{\nu}')[C] - (1-r)\cdot\bar{\nu}'[C] = r\cdot\bar{\mu}'[C].$$

Since $r \neq 0$, it follows $\bar{\mu}[C] = \bar{\mu}'[C]$. Since $\bar{\mu}$ and $\bar{\mu}'$ are stable it follows by Lemma 20 that $\bar{\mu} \leftrightarrows_b \bar{\mu}'$. Consequently, $\mu \leftrightarrows_b \bar{\mu} \leftrightarrows_b \bar{\mu}' \leftrightarrows_b \mu'$. In particular $\mu \leftrightarrows_b \mu'$, as was to be shown. □

# 7 Concluding remarks

We have shown a cancellation law for distributions with respect to branching probabilistic bisimilarity. The result rests on the notion of a stable distribution. Stable distributions enjoy two properties that have been essential to our set-up. (i) Every distribution has a weak unfolding towards a stable distribution that is branching probabilistic bisimilar. (ii) Branching probabilistic bisimilarity for stable distributions is determined by their summed probability for equivalence classes of non-deterministic processes. Techniques from metric topology have been used to establish the first result.

We used the cancellativity result in [16] in order to obtain a complete axiomatisation of branching probabilistic bisimilarity. The technical report [15] contains a proof sketch in line with this paper. Yet, as cancellativity is such a fundamental property, and the notion of branching probabilistic bisimulation is mathematically quite involved, we regard it necessary to provide a full, detailed proof.

The continuity results of Section 5, as well as the argumentation from metric topology at other places, are exploited to deal with the uncountable number of inert transitions that arise from combined transitions. One may wonder if the main theorems of the paper can be achieved based on combinatorial arguments. Intuitively, transitions span a convex polyhedron and the uncountability of the branching of transitions may be reduced to the finiteness of the transitions spanning the polyhedron. Despite a number of attempts, we have been forced to leave the question of a simpler combinatorial proof open.

We leave it as open question for future research weather cancellativity holds for larger classes of probabilistic processes, as could be obtained, for instance, by adding recursion, uncountable choice and/or parallel composition to the syntax. A further topic for future research is the study of cancellativity for other weak variants of probabilistic bisimulation, in particular weak probabilistic bisimulation.

Other future work is to be devoted to the construction of an efficient decision algorithm for branching probabilistic bisimilarity. A decision procedure for strong probabilistic bisimilarity based on so-called extended ordered binary trees has been proposed in [4]. An improved algorithm based on partition refinement is presented in [18]. Partition refinement algorithms for weak and branching probabilistic bisimilarity on states are proposed in [29]. Reduction of weak probabilistic bisimilarity checking of the state-based approach of [8] to linear programming is studied in [12]. Although it is currently not clear how to construct an algorithm deciding branching probabilistic bisimilarity as put forward in this paper, it is likely that the procedures of [17] and [29] can serve as a starting point.

## References


[1] S. Andova, S. Georgievska & N. Trcka (2012): *Branching bisimulation congruence for probabilistic systems*. Theoretical Computer Science 413, pp. 58–72, doi:10.1016/j.tcs.2011.07.020.

[2] S. Andova & T.A.C. Willemse (2006): *Branching bisimulation for probabilistic systems: Characteristics and decidability*. Theoretical Computer Science 356, pp. 325–355, doi:10.1016/j.tcs.2006.02.010.

[3] C. Baier, P.R. D'Argenio & H. Hermanns (2020): *On the probabilistic bisimulation spectrum with silent moves*. Acta Informatica 57, pp. 465–512, doi:10.1007/s00236-020-00379-2.

[4] C. Baier, B. Engelen & M.E. Majster-Cederbaum (2000): *Deciding bisimilarity and similarity for probabilistic processes*. Journal of Computer Systems and Sciences 60(1), pp. 187–231, doi:10.1006/jcss.1999.1683.

[5] C. Baier & M.Z. Kwiatkowska (2000): *Domain equations for probabilistic processes*. Mathematical Structures in Computer Science 10(6), pp. 665–717, doi:10.1017/S0960129599002984.

[6] E. Bandini & R. Segala (2001): *Axiomatizations for Probabilistic Bisimulation*. In F. Orejas et al., editor: Proc. ICALP 2001, LNCS 2076, pp. 370–381, doi:10.1007/3-540-48224-5_31.

[7] F. Breugel & J. Worrell (2005): *A behavioural pseudometric for probabilistic transition systems*. Theoretical Computer Science 331(1), pp. 115–142, doi:10.1016/j.tcs.2004.09.035.

[8] S. Cattani & R. Segala (2002): *Decision Algorithms for Probabilistic Bisimulation*. In L. Brim et al., editor: Proc. CONCUR 2002, LNCS 2421, pp. 371–385, doi:10.1007/3-540-45694-5_25.

[9] Y. Deng, R.J. van Glabbeek, M. Hennessy & C.C. Morgan (2009): *Testing Finitary Probabilistic Processes (extended abstract)*. In M. Bravetti & G. Zavattaro, editors: Proc. CONCUR'09, LNCS 5710, pp. 274–288, doi:10.1007/978-3-642-04081-8_19.

[10] J. Desharnais, V. Gupta, R. Jagadeesan & P. Panangaden (1999): *Metrics for Labeled Markov Systems*. In J.C.M. Baeten & S. Mauw, editors: Proc. CONCUR '99, LNCS 1664, pp. 258–273, doi:10.1007/3-540-48320-9_19.

[11] C. Eisentraut, H. Hermanns, J. Krämer, A. Turrini & L. Zhang (2013): *Deciding Bisimilarities on Distributions*. In K. Joshi et al., editor: Proc. QEST 2013, LNCS 8054, pp. 72–88, doi:10.1007/978-3-642-40196-1_6.

[12] L. Ferrer Fioriti, V. Hashemi, H. Hermanns & A. Turrini (2016): *Deciding probabilistic automata weak bisimulation: theory and practice*. Formal Aspects of Computing 28(1), pp. 109–143, doi:10.1007/s00165-016-0356-4.

[13] A. Giacalone, Chi-Chang Jou & S.A. Smolka (1990): *Algebraic Reasoning for Probabilistic Concurrent Systems*. In M. Broy & C.B. Jones, editors: Programming concepts and methods, North-Holland, pp. 443–458.

[14] M. Giry (1982): *A Categorical Approach to Probability Theory*. In B. Banaschewski, editor: Categorical Aspects of Topology and Analysis, LNM 915, pp. 68–85, doi:10.1007/BFb0092872.

[15] R.J. van Glabbeek, J.F. Groote & E.P. de Vink (2019): *A Complete Axiomatization of Branching Bisimilarity for a Simple Process Language with Probabilistic Choice*. Technical Report, Eindhoven University of Technology. Available at `http://rvg.web.cse.unsw.edu.au/pub/AxiomProbBranchingBis.pdf`.

[16] R.J. van Glabbeek, J.F. Groote & E.P. de Vink (2019): *A Complete Axiomatization of Branching Bisimilarity for a Simple Process Language with Probabilistic Choice (extended abstract)*. In M.A. Alvim et al., editor: The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy, LNCS 11760, pp. 139–162, doi:10.1007/978-3-030-31175-9_9.

[17] J.F. Groote & F.W. Vaandrager (1990): *An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence*. In M. Paterson, editor: Proc. ICALP '90, LNCS 443, pp. 626–638, doi:10.1007/BFb0032063.

[18] J.F. Groote, H.J. Rivera Verduzco & E.P. de Vink (2018): *An efficient algorithm to determine probabilistic bisimulation*. Algorithms 11(9), pp. 131,1–22, doi:10.3390/a11090131.

[19] J.F. Groote & E.P. de Vink (2019): *An Axiomatization of Strong Distribution Bisimulation for a Language with a Parallel Operator and Probabilistic Choice*. In M.H. ter Beek, A. Fantechi & L. Semini, editors: From

*Software Engineering to Formal Methods and Tools, and Back*, LNCS 11865, pp. 449–463, doi:10.1007/978-3-030-30985-5_26.

[20] H. Hansson & B. Jonsson (1990): *A Calculus for Communicating Systems with Time and Probabilities*. In: *Proc. RTSS 1990*, IEEE, pp. 278–287, doi:10.1109/REAL.1990.128759.

[21] J.I. den Hartog, E.P. de Vink & J.W. de Bakker (2000): *Metric semantics and full abstractness for action refinement and probabilistic choice*. *Electronic Notes in Theoretical Computer Science* 40, pp. 72–99, doi:10.1016/S1571-0661(05)80038-6.

[22] M. Hennessy (2012): *Exploring probabilistic bisimulations, part I*. *Formal Aspects of Computing* 24, pp. 749–768, doi:10.1007/s00165-012-0242-7.

[23] S. Lang (1997): *Undergraduate Analysis (2nd ed.)*. Undergraduate Texts in Mathmatics, Springer, doi:10.1007/978-1-4757-2698-5.

[24] M.D. Lee & E.P. de Vink (2016): *Logical Characterization of Bisimulation for Transition Relations over Probability Distributions with Internal Actions*. In P. Faliszewski, A. Muscholl & R. Niedermeier, editors: *Proc. MFCS 2016*, LIPIcs 58, pp. 29:1–29:14, doi:10.4230/LIPIcs.MFCS.2016.29.

[25] G.J. Norman (1997): *Metric Semantics for Probabilistic Systems*. Ph.D. thesis, Universith of Birmingham.

[26] R. Segala (1995): *Modeling and Verification of Randomized Distributed Real-Time Systems*. Ph.D. thesis, MIT. Technical Report MIT/LCS/TR–676.

[27] R. Segala & N.A. Lynch (1994): *Probabilistic simulations for probabilistic processes*. In B. Jonsson & J. Parrow, editors: *Proc. CONCUR 94*, LNCS 836, pp. 481–496, doi:10.1007/978-3-540-48654-1_35.

[28] M. Stoelinga (2002): Alea Jacta est*: Verification of probabilistic, real-time and parametric systems*. Ph.D. thesis, Radboud Universiteit.

[29] A. Turrini & H. Hermanns (2015): *Polynomial time decision algorithms for probabilistic automata*. *Information and Computation* 244, pp. 134–171, doi:10.1016/j.ic.2015.07.004.

# A Lean-Congruence Format for EP-Bisimilarity

Rob van Glabbeek*
School of Informatics
University of Edinburgh, UK
School of Computer Science and Engineering
University of New South Wales
Sydney, Australia
rvg@cs.stanford.edu

Peter Höfner
School of Computing
Australian National University
Canberra, Australia
peter.hoefner@anu.edu.au

Weiyou Wang
weiyou.wang@anu.edu.au

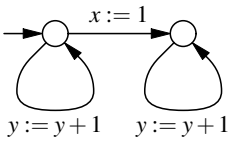
Enabling preserving bisimilarity is a refinement of strong bisimilarity that preserves safety as well as liveness properties. To define it properly, labelled transition systems needed to be upgraded with a successor relation, capturing concurrency between transitions enabled in the same state. We enrich the well-known De Simone format to handle inductive definitions of this successor relation. We then establish that ep-bisimilarity is a congruence for the operators, as well as lean congruence for recursion, for all (enriched) De Simone languages.

## 1 Introduction

Recently, we introduced a finer alternative to strong bisimilarity, called enabling preserving bisimilarity. The motivation behind this concept was to preserve liveness properties, which are *not* always preserved by classical semantic equivalences, including strong bisimilarity.

**Example 1.1 ([13])** Consider the following two programs, and assume that all variables are initialised to 0.

```
while(true) do
  choose
    if true then y := y+1;
    if x = 0 then x := 1;
  end
od
```




```
while(true) do
  y := y+1;
od
||
x := 1;
```

The code on the left-hand side presents a non-terminating while-loop offering an internal nondeterministic choice. The conditional **if** x = 0 then x := 1 describes an atomic read-modify-write operation.[1] Since the non-deterministic choice does not guarantee to ever pick the second conditional, this example should not satisfy the liveness property 'eventually x=1'.

The example on the right-hand side is similar, but here two different components handle the variables x and y separately. The two programs should be considered independent – by default we assume they are executed on different cores. Hence the property 'eventually x=1' should hold.

The two programs behave differently with regards to (some) liveness properties. However, it is easy to verify that they are strongly bisimilar, when considering the traditional modelling of such code in terms of transition systems. In fact, their associated transition systems, also displayed above, are identical. Hence, strong bisimilarity does not preserve all liveness properties.

Enabling preserving bisimilarity (ep-bisimilarity) – see next section for a formal definition – distinguishes these examples and preserves liveness. In contrast to classical bisimulations, which are relations of type States × States, this equivalence is based on triples. An ep-bisimulation additionally maintains

*Supported by Royal Society Wolfson Fellowship RSWF\R1\221008

[1] https://en.wikipedia.org/wiki/Read-modify-write

G. Caltais and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2023).
EPTCS 387, 2023, pp. 59–75, doi:10.4204/EPTCS.387.6

for each pair of related states $p$ and $q$ a relation $R$ between the transitions enabled in $p$ and $q$, and this relation should be preserved when matching related transitions in the bisimulation game. When formalising this, we need transition systems upgraded with a *successor relation* that matches each transition $t$ enabled in a state $p$ to a transition $t'$ enabled in $p'$, when performing a transition from $p$ to $p'$ that does not affect $t$. Intuitively, $t'$ describes the same system behaviour as $t$, but the two transitions could be formally different as they may have different sources. It is this successor relation that distinguishes the transition systems in the example above.

In [13], we showed that ep-bisimilarity is a congruence for all operators of Milner's Calculus of Communication Systems (CCS), enriched with a successor relation. We extended this result to the Algebra of Broadcast Communication with discards and Emissions (ABCdE), an extension of CCS with broadcast communication, discard actions and signal emission. ABCdE subsumes many standard process algebras found in the literature.

In this paper, we introduce a new congruence format for structural operational semantics, which is based on the well-known De Simone Format and respects the successor relation. This format allows us to generalise the results of [13] in two ways: first, we prove that ep-bisimilarity is a congruence for all operators of *any* process algebras that can be formalised in the De Simone format with successors. Applicable languages include CCS and ABCdE. Second, we show that ep-bisimilarity is a lean congruence for recursion [9]. Here, a lean congruence preserves equivalence when replacing closed subexpressions of a process by equivalent alternatives.

## 2 Enabling Preserving Bisimilarity

To build our abstract theory of De Simone languages and De Simone formats, we briefly recapitulate the definitions of labelled transition systems with successors, and ep-bisimulation. A detailed description can be found in [13].

A *labelled transition system (LTS)* is a tuple $(S, Tr, source, target, \ell)$ with $S$ and $Tr$ sets of *states* and *transitions*, $source, target : Tr \to S$ and $\ell : Tr \to \mathscr{L}$, for some set $\mathscr{L}$ of transition labels. A transition $t \in Tr$ of an LTS is *enabled* in a state $p \in S$ if $source(t) = p$. The set of transitions enabled in $p$ is $en(p)$.

**Definition 2.1 (LTSS [13])** A *labelled transition system with successors (LTSS)* is a tuple $(S, Tr, source, target, \ell, \rightsquigarrow)$ with $(S, Tr, source, target, \ell)$ an LTS and $\rightsquigarrow \subseteq Tr \times Tr \times Tr$ the *successor relation* such that if $(t, u, v) \in \rightsquigarrow$ (also denoted by $t \rightsquigarrow_u v$) then $source(t) = source(u)$ and $source(v) = target(u)$.

**Example 2.2** Remember that the 'classical' LTSs of Example 1.1 are identical. Let $t_1$ and $t_2$ be the two transitions corresponding to y:=y+1 in the first and second state, respectively, and let $u$ be the transition for assignment x:=1. The assignments of x and y in the right-hand program are independent, hence $t_1 \rightsquigarrow_u t_2$ and $u \rightsquigarrow_{t_1} u$. For the other program, the situation is different: as the instructions correspond to a single component (program), all transitions affect each other, i.e. $\rightsquigarrow = \emptyset$.

**Definition 2.3 (Ep-bisimilarity [13])** Let $(S, Tr, source, target, \ell, \rightsquigarrow)$ be an LTSS. An *enabling preserving bisimulation (ep-bisimulation)* is a relation $\mathscr{R} \subseteq S \times S \times \mathscr{P}(Tr \times Tr)$ satisfying

1. if $(p, q, R) \in \mathscr{R}$ then $R \subseteq en(p) \times en(q)$ such that
   a. $\forall t \in en(p).\ \exists u \in en(q).\ t\ R\ u$,
   b. $\forall u \in en(q).\ \exists t \in en(p).\ t\ R\ u$, and
   c. if $t\ R\ u$ then $\ell(t) = \ell(u)$; and
2. if $(p, q, R) \in \mathscr{R}$ and $v\ R\ w$, then $(target(v), target(w), R') \in \mathscr{R}$ for some $R'$ such that
   a. if $t\ R\ u$ and $t \rightsquigarrow_v t'$ then $\exists u'.\ u \rightsquigarrow_w u' \wedge t'\ R'\ u'$, and
   b. if $t\ R\ u$ and $u \rightsquigarrow_w u'$ then $\exists t'.\ t \rightsquigarrow_v t' \wedge t'\ R'\ u'$.

Table 1: Structural operational semantics of CCS

$$\frac{}{\alpha.x \xrightarrow{\alpha} x}\ {}^{\alpha}\!\!\rightarrow \qquad \frac{x \xrightarrow{\alpha} x'}{x+y \xrightarrow{\alpha} x'}\ +_{\mathrm{L}} \qquad \frac{y \xrightarrow{\alpha} y'}{x+y \xrightarrow{\alpha} y'}\ +_{\mathrm{R}}$$

$$\frac{x \xrightarrow{\eta} x'}{x|y \xrightarrow{\eta} x'|y}\ |_{\mathrm{L}} \qquad \frac{x \xrightarrow{c} x',\ y \xrightarrow{\bar{c}} y'}{x|y \xrightarrow{\tau} x'|y'}\ |_{\mathrm{C}} \qquad \frac{y \xrightarrow{\eta} y'}{x|y \xrightarrow{\eta} x|y'}\ |_{\mathrm{R}}$$

$$\frac{x \xrightarrow{\ell} x' \quad (\ell \notin L \cup \overline{L})}{x\backslash L \xrightarrow{\ell} x'\backslash L}\ \backslash L \qquad \frac{x \xrightarrow{\ell} x'}{x[f] \xrightarrow{f(\ell)} x'[f]}\ [f] \qquad \frac{\langle S_X|S\rangle \xrightarrow{\alpha} y}{\langle X|S\rangle \xrightarrow{\alpha} y}\ rec_{Act}$$

Two states $p$ and $q$ in an LTSS are *enabling preserving bisimilar (ep-bisimilar)*, denoted as $p \leftrightarrows_{ep} q$, if there is an enabling preserving bisimulation $\mathscr{R}$ such that $(p,q,R) \in \mathscr{R}$ for some $R$.

Without Items 2.a and 2.b, the above is nothing else than a reformulation of the classical definition of strong bisimilarity. An ep-bisimulation additionally maintains for each pair of related states $p$ and $q$ a relation $R$ between the transitions enabled in $p$ and $q$. Items 2.a and 2.b strengthen the condition on related target states by requiring that the successors of related transitions are again related relative to these target states. It is this requirement which distinguishes the transition systems for Example 1.1. [13]

**Lemma 2.4** [Proposition 10 of [13]] $\leftrightarrows_{ep}$ is an equivalence relation.

## 3 An Introductory Example: CCS with Successors

Before starting to introduce the concepts formally, we want to present some motivation in the form of the well-known Calculus of Communicating Systems (CCS) [17]. In this paper we use a proper recursion construct instead of agent identifiers with defining equations. As in [3], we write $\langle X|S\rangle$ for the $X$-component of a solution of the set of recursive equations $S$.

CCS is parametrised with set $\mathscr{C}$ of *handshake communication names*. $\bar{\mathscr{C}} := \{\bar{c} \mid c \in \mathscr{C}\}$ is the set of *handshake communication co-names*. $Act_{CCS} := \mathscr{C} \cup \bar{\mathscr{C}} \cup \{\tau\}$ is the set of *actions*, where $\tau$ is a special *internal action*. Complementation extends to $\mathscr{C} \cup \bar{\mathscr{C}}$ by $\bar{\bar{c}} := c$.

Below, $c$ ranges over $\mathscr{C} \cup \bar{\mathscr{C}}$ and $\alpha$, $\ell$, $\eta$ over $Act_{CCS}$. A *relabelling* is a function $f : \mathscr{C} \to \mathscr{C}$; it extends to $Act_{CCS}$ by $f(\bar{c}) = \overline{f(c)}$, $f(\tau) := \tau$.

The process signature $\Sigma$ of CCS features binary infix-written operators $+$ and $|$, denoting *choice* and *parallel composition*, a constant $\mathbf{0}$ denoting *inaction*, a unary *action prefixing* operator $\alpha._-$ for each action $\alpha \in Act_{CCS}$, a unary *restriction* operator $\_\backslash L$ for each set $L \subseteq \mathscr{C}$, and a unary *relabelling* operator $\_[f]$ for each relabelling $f : \mathscr{C} \to \mathscr{C}$.

The semantics of CCS is given by the set $\mathcal{R}$ of *transition rules*, shown in Table 1. Here $\overline{L} := \{\bar{c} \mid c \in L\}$. Each rule has a unique name, displayed in blue.[2] The rules are displayed as templates, following the standard convention of labelling transitions with *label variables* $c$, $\alpha$, $\ell$, etc. and may be accompanied by side conditions in green, so that each of those templates corresponds to a set of (concrete) transition rules where label variables are "instantiated" to labels in certain ranges and all side conditions are met. The rule names are also schematic and may contain variables. For example, all instances of the transition rule template $+_{\mathrm{L}}$ are named $+_{\mathrm{L}}$, whereas there is one rule name ${}^{\alpha}\!\!\rightarrow$ for each action $\alpha \in Act_{CCS}$.

[2] Our colourings are for readability only.

The transition system specification $(\Sigma,\mathcal{R})$ is in De Simone format [22], a special rule format that guarantees properties of the process algebra (for free), such as strong bisimulation being a congruence for all operators. Following [13], we leave out the infinite sum $\sum_{i\in I} x_i$ of CCS [17], as it is strictly speaking not in De Simone format.

In this paper, we will extend the De Simone format to also guarantee properties for ep-bisimulation. As seen, ep-bisimulation requires that the structural operational semantics is equipped with a successor relation $\rightsquigarrow$. The meaning of $\chi \rightsquigarrow_\zeta \chi'$ is that transition $\chi$ is unaffected by $\zeta$ – denoted $\chi \frown \zeta$ – and that when doing $\zeta$ instead of $\chi$, afterwards a variant $\chi'$ of $\chi$ is still enabled. Table 2 shows the *successor rules* for CCS, which allow the relation $\rightsquigarrow$ to be derived inductively. It uses the following syntax for transitions $\chi$, which will be formally introduced in Section 6. The expression $t{+_{\mathrm{L}}}Q$ refers to the transition that is derived by rule ${+_{\mathrm{L}}}$ of Table 1, with $t$ referring to the transition used in the unique premise of this rule, and $Q$ referring to the process in the inactive argument of the $+$-operator. The syntax for the other transitions is analogous. A small deviation of this scheme occurs for recursion: $rec_{Act}(X,S,t)$ refers to the transition derived by rule $rec_{Act}$ out of the premise $t$, when deriving a transition of a recursive call $\langle X|S\rangle$.

In Table 2 each rule is named, in orange, after the number of the clause of Definition 20 in [13], were it was introduced.

The primary source of concurrency between transition $\chi$ and $\zeta$ is when they stem from opposite sides of a parallel composition. This is expressed by Rules 7a and 7b. We require all obtained successor statements $\chi \rightsquigarrow_\zeta \chi'$ to satisfy the conditions of Definition 2.1 – this yields $Q' = target(w)$ and $P' = target(v)$; in [13] $Q'$ and $P'$ were written this way.

In all other cases, successors of $\chi$ are inherited from successors of their building blocks.

When $\zeta$ stems from the left side of a $+$ via rule ${+_{\mathrm{L}}}$ of Table 1, then any transition $\chi$ stemming from the right is discarded by $\zeta$, so $\chi \not\frown \zeta$. Thus, if $\chi \frown \zeta$ then these transitions have the form $\chi = t{+_{\mathrm{L}}}Q$ and $\zeta = v{+_{\mathrm{L}}}Q$, and we must have $t \frown v$. So $t \rightsquigarrow_v t'$ for some transition $t'$. As the execution of $\zeta$ discards the summand $Q$, we also obtain $\chi \rightsquigarrow_\zeta t'$. This motivates Rule 3a. Rule 4a follows by symmetry.

In a similar way, Rule 8a covers the case that $\chi$ and $\zeta$ both stem from the left component of a parallel composition. It can also happen that $\chi$ stems form the left component, whereas $\zeta$ is a synchronisation, involving both components. Thus $\chi = t|_{\mathrm{L}}Q$ and $\zeta = v|_{\mathrm{C}}w$. For $\chi \frown \zeta$ to hold, it must be that $t \frown v$, whereas the $w$-part of $\zeta$ cannot interfere with $t$. This yields the Rule 8b. Rule 8c is explained in a similar vain from the possibility that $\zeta$ stems from the left while $\chi$ is a synchronisation of both components. Rule 9 follows by symmetry. In case both $\chi$ and $\zeta$ are synchronisations involving both components, i.e., $\chi = t|_{\mathrm{C}}u$ and $\zeta = v|_{\mathrm{C}}w$, it must be that $t \frown v$ and $u \frown w$. Now the resulting variant $\chi'$ of $\chi$ after $\zeta$ is simply $t'|u'$, where $t \rightsquigarrow_v t'$ and $u \rightsquigarrow_w u'$. This underpins Rule 10.

If the common source $O$ of $\chi$ and $\zeta$ has the form $P[f]$, $\chi$ and $\zeta$ must have the form $t[f]$ and $v[f]$. Whether $t$ and $v$ are concurrent is not influenced by the renaming. So $t \frown v$. The variant of $t$ that remains after doing $v$ is also not affected by the renaming, so if $t \rightsquigarrow_v t'$ then $\chi \rightsquigarrow_\zeta t'[f]$. The case that $O = P\backslash L$ is equally trivial. This yields Rules 11a and 11b.

In case $O = \langle X|S\rangle$, $\chi$ must have the form $rec_{Act}(X,S,t)$, and $\zeta$ has the form $rec_{Act}(X,S,v)$, where $t$ and $v$ are enabled in $\langle S_X|S\rangle$. Now $\chi \frown \zeta$ only if $t \frown v$, so $t \rightsquigarrow_v t'$ for some transition $t'$. The recursive call disappears upon executing $\zeta$, and we obtain $\chi \rightsquigarrow_\zeta t'$. This yields Rule 11c.

**Example 3.1** The programs from Example 1.1 could be represented in CCS as $P := \langle X|S\rangle$ where $S = \left\{ \begin{array}{l} X = a.X + b.Y \\ Y = a.Y \end{array} \right\}$ and $Q := \langle Z|\{Z = a.Z\}\rangle|b.\mathbf{0}$. Here $a,b \in Act_{CCS}$ are the atomic actions incrementing $y$ and $x$. The relation matching $P$ with $Q$ and $\langle Y,S\rangle$ with $\langle Z|\{Z = a.Z\}\rangle|\mathbf{0}$ is a strong bisimulation. Yet, $P$ and $Q$ are not ep-bisimilar, as the rules of Table 2 derive $u \rightsquigarrow_{t_1} u$ (cf. Example 2.2)

Table 2: Successor rules for CCS

$$\frac{t \rightsquigarrow_v t'}{t +_{\mathrm{L}} Q \rightsquigarrow_{v +_{\mathrm{L}} Q} t'}\ 3a \qquad \frac{u \rightsquigarrow_w u'}{P +_{\mathrm{R}} u \rightsquigarrow_{P +_{\mathrm{R}} w} u'}\ 4a$$

$$\frac{}{t|_{\mathrm{L}} Q \rightsquigarrow_{P|_{\mathrm{R}} w} t|_{\mathrm{L}} Q'}\ 7a \qquad \frac{t \rightsquigarrow_v t' \quad u \rightsquigarrow_w u'}{t|_{\mathrm{C}} u \rightsquigarrow_{v|_{\mathrm{C}} w} t'|_{\mathrm{C}} u'}\ 10 \qquad \frac{}{P|_{\mathrm{R}} u \rightsquigarrow_{v|_{\mathrm{L}} Q} P'|_{\mathrm{R}} u}\ 7b$$

$$\frac{t \rightsquigarrow_v t'}{t|_{\mathrm{L}} Q \rightsquigarrow_{v|_{\mathrm{L}} Q} t'|_{\mathrm{L}} Q}\ 8a \qquad \frac{t \rightsquigarrow_v t'}{t|_{\mathrm{L}} Q \rightsquigarrow_{v|_{\mathrm{C}} w} t'|_{\mathrm{L}} Q'}\ 8b \qquad \frac{t \rightsquigarrow_v t'}{t|_{\mathrm{C}} u \rightsquigarrow_{v|_{\mathrm{L}} Q} t'|_{\mathrm{C}} u}\ 8c$$

$$\frac{u \rightsquigarrow_w u'}{P|_{\mathrm{R}} u \rightsquigarrow_{P|_{\mathrm{R}} w} P|_{\mathrm{R}} u'}\ 9a \qquad \frac{u \rightsquigarrow_w u'}{P|_{\mathrm{R}} u \rightsquigarrow_{v|_{\mathrm{C}} w} P'|_{\mathrm{R}} u'}\ 9b \qquad \frac{u \rightsquigarrow_w u'}{t|_{\mathrm{C}} u \rightsquigarrow_{P|_{\mathrm{R}} w} t|_{\mathrm{C}} u'}\ 9c$$

$$\frac{t \rightsquigarrow_v t'}{t\backslash L \rightsquigarrow_{v\backslash L} t'\backslash L}\ 11a \qquad \frac{t \rightsquigarrow_v t'}{t[f] \rightsquigarrow_{v[f]} t'[f]}\ 11b \qquad \frac{t \rightsquigarrow_v t'}{rec_{Act}(X,S,t) \rightsquigarrow_{rec_{Act}(X,S,v)} t'}\ 11c$$

where $u = \langle Z | \{Z = a.Z\}\rangle|_{\mathrm{R}} \xrightarrow{b} \mathbf{0}$ and $t_1 = rec_{Act}(Z, \{Z{=}a.Z\}, \xrightarrow{a} Q)|_{\mathrm{L}} b.\mathbf{0}$. This cannot be matched by $P$, thus violating condition 2.b. of Definition 2.3.

In this paper we will introduce a new De Simone format for transition systems with successors (TSSS). We will show that $\leftrightarrows_{ep}$ is a congruence for all operators (as well as a lean congruence for recursion) in any language that fits this format. Since the rules of Table 2 fit this new De Simone format, it follows that $\leftrightarrows_{ep}$ is a congruence for the operators of CCS.

Informally, the conclusion of a successor rule in this extension of the De Simone format must have the form $\zeta \rightsquigarrow_\xi \zeta'$ where $\zeta$, $\xi$ and $\zeta'$ are *open transitions*, denoted by *transition expressions* with variables, formally introduced in Section 6. Both $\zeta$ and $\xi$ must have a leading operator $\mathrm{R}$ and $\mathrm{S}$ of the same type, and the same number of arguments. These leading operators must be rule names of the same type. Their arguments are either process variables $P, Q, \ldots$ or transition variables $t, u, \ldots$, as determined by the trigger sets $I_{\mathrm{R}}$ and $I_{\mathrm{S}}$ of $\mathrm{R}$ and $\mathrm{S}$. These are the sets of indices listing the arguments for which rules $\mathrm{R}$ and $\mathrm{S}$ have a premise. If the $i^{\text{th}}$ arguments of $\mathrm{R}$ and $\mathrm{S}$ are both process variables, they must be the same, but for the rest all these variables are different. For a subset $I$ of $I_{\mathrm{R}} \cap I_{\mathrm{S}}$, the rule has premises $t_i \rightsquigarrow_{u_i} t'_i$ for $i \in I$, where $t_i$ and $u_i$ are the $i^{\text{th}}$ arguments of $\mathrm{R}$ and $\mathrm{S}$, and $t'_i$ is a fresh variable. Finally, the right-hand side of the conclusion may be an arbitrary univariate transition expression, containing no other variables than:

- the $t'_i$ for $i \in I$,
- a $t_i$ occurring in $\zeta$, with $i \notin I_{\mathrm{S}}$,
- a fresh process variable $P'_i$ that must match the target of the transition $u_i$ for $i \in I_{\mathrm{S}} \backslash I$,
- *or* a fresh transition variable whose source matches the target of $u_i$ for $i \in I_{\mathrm{S}} \backslash I$, and
- any $P$ occurring in both $\zeta$ and $\xi$, *or* any fresh transition variable whose source must be $P$.

The rules of Table 2 only feature the first three possibilities; the others occur in the successor relation of ABCdE – see Section 8.

## 4 Structural Operational Semantics

Both the De Simone format and our forthcoming extension are based on the syntactic form of the operational rules. In this section, we recapitulate foundational definitions needed later on. Let $\mathcal{V}_{\mathcal{P}}$ be an infinite set of *process variables*, ranged over by $X, Y, x, y, x_i$, etc.

**Definition 4.1 (Process Expressions [8])** An *operator declaration* is a pair $(Op,n)$ of an *operator symbol* $Op \notin \mathcal{V}_\mathcal{P}$ and an *arity* $n \in \mathbb{N}$. An operator declaration $(c,0)$ is also called a *constant declaration*. A *process signature* is a set of operator declarations. The set $\mathbb{P}^r(\Sigma)$ of *process expressions* over a process signature $\Sigma$ is defined inductively by:

- $\mathcal{V}_\mathcal{P} \subseteq \mathbb{P}^r(\Sigma)$,
- if $(Op,n) \in \Sigma$ and $p_1,\ldots,p_n \in \mathbb{P}^r(\Sigma)$ then $Op(p_1,\ldots,p_n) \in \mathbb{P}^r(\Sigma)$, and
- if $V_S \subseteq \mathcal{V}_\mathcal{P}$, $S : V_S \to \mathbb{P}^r(\Sigma)$ and $X \in V_S$, then $\langle X|S\rangle \in \mathbb{P}^r(\Sigma)$.

A process expression $c()$ is abbreviated as $c$ and is also called a *constant*. An expression $\langle X|S\rangle$ as appears in the last clause is called a *recursive call*, and the function $S$ therein is called a *recursive specification*. It is often displayed as $\{X = S_X \mid X \in V_S\}$. Therefore, for a recursive specification $S$, $V_S$ denotes the domain of $S$ and $S_X$ represents $S(X)$ when $X \in V_S$. Each expression $S_Y$ for $Y \in V_S$ counts as a subexpression of $\langle X|S\rangle$. An occurrence of a process variable $y$ in an expression $p$ is *free* if it does not occur in a subexpression of the form $\langle X|S\rangle$ with $y \in V_S$. For an expression $p$, $var(p)$ denotes the set of process variables having at least one free occurrence in $p$. An expression is *closed* if it contains no free occurrences of variables. Let $\mathrm{P}^r(\Sigma)$ be the set of closed process expressions over $\Sigma$.

**Definition 4.2 (Substitution)** A $\Sigma$-*substitution* $\sigma$ is a partial function from $\mathcal{V}_\mathcal{P}$ to $\mathbb{P}^r(\Sigma)$. It is *closed* if it is a total function from $\mathcal{V}_\mathcal{P}$ to $\mathrm{P}^r(\Sigma)$.

If $p \in \mathbb{P}^r(\Sigma)$ and $\sigma$ a $\Sigma$-substitution, then $p[\sigma]$ denotes the expression obtained from $p$ by replacing, for $x$ in the domain of $\sigma$, every free occurrence of $x$ in $p$ by $\sigma(x)$, while renaming bound process variables if necessary to prevent name-clashes. In that case $p[\sigma]$ is called a *substitution instance* of $p$. A substitution instance $p[\sigma]$ where $\sigma$ is given by $\sigma(x_i) = q_i$ for $i \in I$ is denoted as $p[q_i/x_i]_{i\in I}$, and for $S$ a recursive specification $\langle p|S\rangle$ abbreviates $p[\langle Y|S\rangle/Y]_{Y\in V_S}$.

These notions, including "free" and "closed", extend to syntactic objects containing expressions, with the understanding that such an object is a substitution instance of another one if the same substitution has been applied to each of its constituent expressions.

We assume fixed but arbitrary sets $\mathscr{L}$ and $\mathcal{N}$ of *transition labels* and *rule names*.

**Definition 4.3 (Transition System Specification [16])** Let $\Sigma$ be a process signature. A $\Sigma$-*(transition) literal* is an expression $p \xrightarrow{a} q$ with $p,q \in \mathbb{P}^r(\Sigma)$ and $a \in \mathscr{L}$. A *transition rule* over $\Sigma$ is an expression of the form $\frac{H}{\lambda}$ with $H$ a finite list of $\Sigma$-literals (the *premises* of the transition rule) and $\lambda$ a $\Sigma$-literal (the *conclusion*). A *transition system specification (TSS)* is a tuple $(\Sigma,\mathcal{R},\mathrm{N})$ with $\mathcal{R}$ a set of transition rules over $\Sigma$, and $\mathrm{N} : \mathcal{R} \to \mathcal{N}$ a (not necessarily injective) *rule-naming function*, that provides each rule $r \in \mathcal{R}$ with a name $\mathrm{N}(r)$.

**Definition 4.4 (Proof)** Assume literals, rules, substitution instances and rule-naming. A *proof* of a literal $\lambda$ from a set $\mathcal{R}$ of rules is a well-founded, upwardly branching, ordered tree where nodes are labelled by pairs $(\mu,\mathrm{R})$ of a literal $\mu$ and a rule name $\mathrm{R}$, such that

- the root is labelled by a pair $(\lambda,\mathrm{S})$, and
- if $(\mu,\mathrm{R})$ is the label of a node and $(\mu_1,\mathrm{R}_1),\ldots,(\mu_n,\mathrm{R}_n)$ is the list of labels of this node's children then $\frac{\mu_1,\ldots,\mu_n}{\mu}$ is a substitution instance of a rule in $\mathcal{R}$ with name $\mathrm{R}$.

**Definition 4.5 (Associated LTS [12])** The *associated LTS* of a TSS $(\Sigma,\mathcal{R},\mathrm{N})$ is the LTS $(S,Tr,source,target,\ell)$ with $S := \mathrm{P}^r(\Sigma)$ and $Tr$ the collection of proofs $\pi$ of closed $\Sigma$-literals $p \xrightarrow{a} q$ from $\mathcal{R}$, where $source(\pi) = p$, $\ell(\pi) = a$ and $target(\pi) = q$.

Above we deviate from the standard treatment of structural operational semantics [16, 8] on four counts. Here we employ CCS to motivate those design decisions.

In Definition 4.5, the transitions $Tr$ are taken to be proofs of closed literals $p \xrightarrow{a} q$ rather than such literals themselves. This is because there can be multiple $a$-transitions from $p$ to $q$ that need to be distinguished when taking the concurrency relation between transitions into account. For example, if $p := \langle X | \{X = a.X + c.X\}\rangle$ and $q := \langle Y | \{Y = a.Y\}\rangle$ then $p|q$ has three outgoing transitions:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a.p \xrightarrow{a} p}\,{\scriptstyle \xrightarrow{a}}}{a.p + c.p \xrightarrow{a} p}\,{\scriptstyle +_{\mathrm{L}}}}{p \xrightarrow{a} p}\,{\scriptstyle rec_{Act}}}{p|q \xrightarrow{a} p|q}\,{\scriptstyle |_{\mathrm{L}}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{c.p \xrightarrow{c} p}\,{\scriptstyle \xrightarrow{c}}}{a.p + c.p \xrightarrow{c} p}\,{\scriptstyle +_{\mathrm{R}}}}{p \xrightarrow{c} p}\,{\scriptstyle rec_{Act}}}{p|q \xrightarrow{c} p|q}\,{\scriptstyle |_{\mathrm{L}}} \qquad \dfrac{\dfrac{\dfrac{}{a.q \xrightarrow{a} q}\,{\scriptstyle \xrightarrow{a}}}{q \xrightarrow{a} q}\,{\scriptstyle rec_{Act}}}{p|q \xrightarrow{a} p|q}\,{\scriptstyle |_{\mathrm{R}}}$$

The rightmost transition is concurrent with the middle one, whereas the leftmost one is not.

A similar example can be used to motivate why in Definition 4.4 the nodes are labelled not only by the inferred literal, but also by the name of the applied rule.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a.p \xrightarrow{a} p}\,{\scriptstyle \xrightarrow{a}}}{a.p + c.p \xrightarrow{a} p}\,{\scriptstyle +_{\mathrm{L}}}}{p \xrightarrow{a} p}\,{\scriptstyle rec_{Act}}}{p|p \xrightarrow{a} p|p}\,{\scriptstyle |_{\mathrm{L}}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{c.p \xrightarrow{c} p}\,{\scriptstyle \xrightarrow{c}}}{a.p + c.p \xrightarrow{c} p}\,{\scriptstyle +_{\mathrm{R}}}}{p \xrightarrow{c} p}\,{\scriptstyle rec_{Act}}}{p|p \xrightarrow{c} p|p}\,{\scriptstyle |_{\mathrm{L}}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{a.p \xrightarrow{a} p}\,{\scriptstyle \xrightarrow{a}}}{a.p + c.p \xrightarrow{a} p}\,{\scriptstyle +_{\mathrm{L}}}}{p \xrightarrow{a} p}\,{\scriptstyle rec_{Act}}}{p|p \xrightarrow{a} p|p}\,{\scriptstyle |_{\mathrm{R}}}$$

The rightmost transition is concurrent with the middle one, but the leftmost one is not. If we were to erase the rule names, the difference between these two transitions would disappear.

In Definition 4.3 we require the premises of rules to be lists rather than sets, and accordingly in Definition 4.4 we require proof trees to be ordered. This is to distinguish transitions/proofs in which a substitution instance of a rule has two identical premises (corresponding to different arguments of the leading operator) with different proofs. This phenomenon does not occur in CCS, but we could have illustrated it with CSP [5] or ABCdE [13].

Finally, suppose that in Definition 4.3 we had chosen the rule-naming function $\mathrm{N}$ to be the identity. This is equivalent to not having a rule-naming function at all, instead labelling nodes in proofs with rules rather than names of rules. Then in the transition

$$\dfrac{\dfrac{\dfrac{}{a.\mathbf{0} \xrightarrow{a} \mathbf{0}}\,{\scriptstyle \xrightarrow{a}}}{\langle X | \{X = a.\mathbf{0}\}\rangle \xrightarrow{a} \mathbf{0}}\,{\scriptstyle rec_{Act}}$$

we should replace the generic name $rec_{Act}$ of a recursion rule with the specific rule employed. This could be the rule $\dfrac{a.\mathbf{0} \xrightarrow{a} z}{\langle X | \{X = a.\mathbf{0}\}\rangle \xrightarrow{a} z}$, but just as well the rule $\dfrac{y \xrightarrow{a} z}{\langle X | \{X = y\}\rangle \xrightarrow{a} z}$, when employing a substitution that sends $y$ to $a.\mathbf{0}$. To avoid the resulting unnecessary duplication of transitions, we give both recursion rules the same name.

## 5 De Simone Languages

The syntax of a *De Simone language* is specified by a process signature, and its semantics is given as a TSS over that process signature of a particular form [22], nowadays known as the *De Simone format*. Here, we extend the De Simone format to support *indicator transitions*, as occur in [11, 10, 13]. These are transitions $p \xrightarrow{\ell} q$ for which it is essential that $p = q$. They are used to convey a property of the state $p$ rather than model an action of $p$. To accommodate them we need a variant of the recursion rule whose conclusion again is of the form $r \xrightarrow{\ell} r$. This variant will be illustrated in Section 8.

As for $\mathscr{L}$, we fix a set $Act \subseteq \mathscr{L}$ of *actions*.

**Definition 5.1 (De Simone Format)** A TSS $(\Sigma, \mathcal{R}, \mathrm{N})$ is in *De Simone format* if for every recursive call $\langle X|S\rangle$ and every $\alpha \in Act$ and $\ell \in \mathscr{L}\backslash Act$, it has transition rules

$$\frac{\langle S_X|S\rangle \xrightarrow{\alpha} y}{\langle X|S\rangle \xrightarrow{\alpha} y}\ rec_{Act} \qquad \text{and} \qquad \frac{\langle S_X|S\rangle \xrightarrow{\ell} y}{\langle X|S\rangle \xrightarrow{\ell} \langle X|S\rangle}\ rec_{In} \quad \text{for some} \quad y \notin var(\langle S_X|S\rangle),$$

and each of its other transition rules (*De Simone rules*) has the form

$$\frac{\{x_i \xrightarrow{a_i} y_i \mid i \in I\}}{Op(x_1,\ldots,x_n) \xrightarrow{a} q}$$

where $(Op,n) \in \Sigma$, $I \subseteq \{1,\ldots,n\}$, $a, a_i \in \mathscr{L}$, $x_i$ (for $1 \le i \le n$) and $y_i$ (for $i \in I$) are pairwise distinct process variables, and $q$ is a univariate process expression containing no other free process variables than $x_i$ $(1 \le i \le n \wedge i \notin I)$ and $y_i$ $(i \in I)$, having the properties that

- each subexpression of the form $\langle X|S\rangle$ is closed, and
- if $a \in \mathscr{L}\backslash Act$ then $a_i \in \mathscr{L}\backslash Act$ $(i \in I)$ and $q = Op(z_1,\ldots,z_n)$, where $z_i := \begin{cases} y_i & \text{if } i \in I \\ x_i & \text{otherwise.} \end{cases}$

Here *univariate* means that each variable has at most one free occurrence in it. The last clause above guarantees that for any indicator transition $t$, one with $\ell(t) \in \mathscr{L}\backslash Act$, we have $target(t) = source(t)$. For a De Simone rule of the above form, $n$ is the *arity*, $(Op,n)$ is the *type*, $a$ is the *label*, $q$ is the *target*, $I$ is the *trigger set* and the tuple $(\ell_i,\ldots,\ell_n)$ with $\ell_i = a_i$ if $i \in I$ and $\ell_i = *$ otherwise, is the *trigger*. Transition rules in the first two clauses are called *recursion rules*.

We also require that if $\mathrm{N}(r) = \mathrm{N}(r')$ for two different De Simone rules $r, r' \in \mathcal{R}$, then $r, r'$ have the same type, target and trigger set, but different triggers. The names of the recursion rules are as indicated in blue above, and differ from the names of any De Simone rules.

Many process description languages encountered in the literature, including CCS [17] as presented in Section 3, SCCS [18], ACP [3] and MEIJE [2], are De Simone languages.

## 6 Transition System Specifications with Successors

In Section 4, a *process* is denoted by a closed process expression; an open process expression may contain variables, which stand for as-of-yet unspecified subprocesses. Here we will do the same for transition expressions with variables. However, in this paper a transition is defined as a proof of a literal $p \xrightarrow{a} q$ from the operational rules of a language. Elsewhere, a transition is often defined as a provable literal $p \xrightarrow{a} q$, but here we need to distinguish transitions based on these proofs, as this influences whether two transitions are concurrent.

It turns out to be convenient to introduce an *open proof* of a literal as the semantic interpretation of an open transition expression. It is simply a proof in which certain subproofs are replaced by proof variables.

**Definition 6.1 (Open Proof)** Given definitions of literals, rules and substitution instances, and a rule-naming function N, an *open proof* of a literal $\lambda$ from a set $\mathcal{R}$ of rules using a set $\mathcal{V}$ of *(proof) variables* is a well-founded, upwardly branching, ordered tree of which the nodes are labelled either by pairs $(\mu, \mathrm{R})$ of a literal $\mu$ and a rule name R, or by pairs $(\mu, px)$ of a literal $\mu$ and a variable $px \in \mathcal{V}$ such that

- the root is labelled by a pair $(\lambda, \chi)$,
- if $(\mu, px)$ is the label of a node then this node has no children,

- if two nodes are labelled by $(\mu, px)$ and $(\mu', px)$ separately then $\mu = \mu'$, and
- if $(\mu, \mathrm{R})$ is the label of a node and $(\mu_1, \chi_1), \ldots, (\mu_n, \chi_n)$ is the list of labels of this node's children then $\frac{\mu_1, \ldots, \mu_n}{\mu}$ is a substitution instance of a rule named R.

Let $\mathcal{V}_\mathcal{T}$ be an infinite set of *transition variables*, disjoint from $\mathcal{V}_\mathcal{P}$. We will use $tx, ux, vx, ty, tx_i$, etc. to range over $\mathcal{V}_\mathcal{T}$.

**Definition 6.2 (Open Transition)** Fix a TSS $(\Sigma, \mathcal{R}, \mathrm{N})$. An *open transition* is an open proof of a $\Sigma$-literal from $\mathcal{R}$ using $\mathcal{V}_\mathcal{T}$. For an open transition $\mathring{t}$, $var_\mathcal{T}(\mathring{t})$ denotes the set of transition variables occurring in $\mathring{t}$; if its root is labelled by $(p \xrightarrow{a} q, \chi)$ then $src_\circ(\mathring{t}) = p$, $\ell_\circ(\mathring{t}) = a$ and $tar_\circ(\mathring{t}) = q$. The *binding function* $\beta_{\mathring{t}}$ of $\mathring{t}$ from $var_\mathcal{T}(\mathring{t})$ to $\Sigma$-literals is defined by $\beta_{\mathring{t}}(tx) = \mu$ if $tx \in var_\mathcal{T}(\mathring{t})$ and $(\mu, tx)$ is the label of a node in $\mathring{t}$. Given an open transition, we refer to the subproofs obtained by deleting the root node as its *direct subtransitions*.

All occurrences of transition variables are considered *free*. Let $\mathbb{T}^r(\Sigma, \mathcal{R}, \mathrm{N})$ be the set of open transitions in the TSS $(\Sigma, \mathcal{R}, \mathrm{N})$ and $\mathrm{T}^r(\Sigma, \mathcal{R}, \mathrm{N})$ the set of closed open transitions. We have $\mathrm{T}^r(\Sigma, \mathcal{R}, \mathrm{N}) = Tr$.

Let $en_\circ(p)$ denote $\{\mathring{t} \mid src_\circ(\mathring{t}) = p\}$.

**Definition 6.3 (Transition Expression)** A *transition declaration* is a tuple $(\mathrm{R}, n, I)$ of a *transition constructor* R, an arity $n \in \mathbb{N}$ and a trigger set $I \subseteq \{1, \ldots, n\}$. A *transition signature* is a set of transition declarations. The set $\mathbb{TE}^r(\Sigma_\mathcal{P}, \Sigma_\mathcal{T})$ of *transition expressions* over a process signature $\Sigma_\mathcal{P}$ and a transition signature $\Sigma_\mathcal{T}$ is defined inductively as follows.

- if $tx \in \mathcal{V}_\mathcal{T}$ and $\mu$ is a $\Sigma$-literal then $(tx :: \mu) \in \mathbb{TE}^r(\Sigma_\mathcal{P}, \Sigma_\mathcal{T})$,
- if $E \in \mathbb{TE}^r(\Sigma_\mathcal{P}, \Sigma_\mathcal{T})$, $S : \mathcal{V}_\mathcal{P} \rightharpoonup \mathbb{P}^r(\Sigma_\mathcal{P})$ and $X \in \mathrm{dom}(S)$ then $rec_{Act}(X, S, E), rec_{In}(X, S, E) \in \mathbb{TE}^r(\Sigma_\mathcal{P}, \Sigma_\mathcal{T})$, and
- if $(\mathrm{R}, n, I) \in \Sigma_\mathcal{T}$, $E_i \in \mathbb{TE}^r(\Sigma_\mathcal{P}, \Sigma_\mathcal{T})$ for each $i \in I$, and $E_i \in \mathbb{P}^r(\Sigma_\mathcal{P})$ for each $i \in \{1, \ldots, n\} \setminus I$, then $\mathrm{R}(E_1, \ldots, E_n) \in \mathbb{TE}^r(\Sigma_\mathcal{P}, \Sigma_\mathcal{T})$.

Given a TSS $(\Sigma, \mathcal{R}, \mathrm{N})$ in De Simone format, each open transition $\mathring{t} \in \mathbb{T}^r(\Sigma, \mathcal{R})$ is named by a unique transition expression in $\mathbb{TE}^r(\Sigma, \Sigma_\mathcal{T})$; here $\Sigma_\mathcal{T} = \{(\mathrm{N}(r), n, I) \mid r \in \mathcal{R}$ is a De Simone rule, $n$ is its arity and $I$ is its trigger set$\}$:

- if the root of $\mathring{t}$ is labelled by $(\mu, tx)$ where $tx \in \mathcal{V}_\mathcal{T}$ then $\mathring{t}$ is named $(tx :: \mu)$,
- if the root of $\mathring{t}$ is labelled by $(\langle X|S\rangle \xrightarrow{a} q, \mathrm{R})$ where $a \in Act$ then $\mathring{t}$ is named $rec_{Act}(X, S, E)$ where $E$ is the name of the direct subtransition of $\mathring{t}$,
- if the root of $\mathring{t}$ is labelled by $(\langle X|S\rangle \xrightarrow{\ell} \langle X|S\rangle, \mathrm{R})$ where $\ell \in \mathscr{L} \setminus Act$ then $\mathring{t}$ is named $rec_{In}(X, S, E)$ where $E$ is the name of the direct subtransition of $\mathring{t}$, and
- if the root of $\mathring{t}$ is labelled by $(Op(p_1, \ldots, p_n) \xrightarrow{a} q, \mathrm{R})$ then $\mathring{t}$ is named $\mathrm{R}(E_1, \ldots, E_n)$ where, letting $n$ and $I$ be the arity and the trigger set of the rules named R, $E_i$ for each $i \in I$ is the name of the direct subtransitions of $\mathring{t}$ corresponding to the index $i$, and $E_i = p_i$ for each $i \in \{1, \ldots, n\} \setminus I$.

We now see that the first requirement for the rule-naming function in Definition 5.1 ensures that every open transition is uniquely identified by its name.

**Definition 6.4 (Transition Substitution)** Let $(\Sigma, \mathcal{R}, \mathrm{N})$ be a TSS. A $(\Sigma, \mathcal{R})$-*substitution* is a partial function $\sigma_\mathcal{T} : (\mathcal{V}_\mathcal{P} \rightharpoonup \mathbb{P}^r(\Sigma)) \cup (\mathcal{V}_\mathcal{T} \rightharpoonup \mathbb{T}^r(\Sigma, \mathcal{R}))$. It is *closed* if it is a total function $\sigma_\mathcal{T} : (\mathcal{V}_\mathcal{P} \to \mathrm{P}^r(\Sigma)) \cup (\mathcal{V}_\mathcal{T} \to \mathrm{T}^r(\Sigma, \mathcal{R}))$. A $(\Sigma, \mathcal{R})$-substitution $\sigma_\mathcal{T}$ *matches* all process expressions. It matches an open transition $\mathring{t}$ whose binding function is $\beta_{\mathring{t}}$ if for all $(tx, \mu) \in \beta_{\mathring{t}}$, $\sigma_\mathcal{T}(tx)$ being defined and $\mu = (p \xrightarrow{a} q)$ implies $\ell_\circ(\sigma_\mathcal{T}(tx)) = a$ and $src_\circ(\sigma_\mathcal{T}(tx)), tar_\circ(\sigma_\mathcal{T}(tx))$ being the substitution instances of $p, q$ respectively by applying $\sigma_\mathcal{T} \restriction \mathcal{V}_\mathcal{P}$.

If $E \in \mathbb{P}^r(\Sigma) \cup \mathbb{T}^r(\Sigma, \mathcal{R})$ and $\sigma_\mathcal{T}$ is a $(\Sigma, \mathcal{R})$-substitution matching $E$, then $E[\sigma_\mathcal{T}]$ denotes the expression obtained from $E$ by replacing, for $tx \in \mathcal{V}_\mathcal{T}$ in the domain of $\sigma_\mathcal{T}$, every subexpression of the form

$(tx :: \mu)$ in $E$ by $\sigma_{\mathcal{T}}(tx)$, and for $x \in \mathcal{V}_{\mathcal{P}}$ in the domain of $\sigma_{\mathcal{T}}$, every free occurrence of $x$ in $E$ by $\sigma_{\mathcal{T}}(x)$, while renaming bound process variables if necessary to prevent name-clashes. In that case $E[\sigma_{\mathcal{T}}]$ is called a *substitution instance* of $E$.

Note that a substitution instance of an open transition can be a transition expression not representing an open transition. For example, applying a $(\Sigma,\mathcal{R})$-substitution $\sigma_{\mathcal{T}}$ given by $\sigma_{\mathcal{T}}(ty) := (tx :: y \xrightarrow{\bar{c}} y')$ to the open transition $(tx :: x \xrightarrow{c} x') \mid (ty :: y \xrightarrow{c} y')$ results in $(tx :: x \xrightarrow{c} x') \mid (tx :: y \xrightarrow{c} y')$ which is not an open transition because the transition variable $tx$ is used for two different $\Sigma$-literals. This will not happen if $\sigma_{\mathcal{T}}$ is closed.

**Observation 6.5** Given a TSS $(\Sigma,\mathcal{R},\mathrm{N})$, if $\mathring{t} \in en_{\circ}(p)$ is a open transition and $\sigma_{\mathcal{T}}$ is a closed $(\Sigma,\mathcal{R})$-substitution which matches $\mathring{t}$ then $\mathring{t}[\sigma_{\mathcal{T}}] \in Tr$, $source(\mathring{t}[\sigma_{\mathcal{T}}]) = src_{\circ}(\mathring{t})[\sigma_{\mathcal{T}}]$, $\ell(\mathring{t}[\sigma_{\mathcal{T}}]) = \ell_{\circ}(\mathring{t})$ and $target(\mathring{t}[\sigma_{\mathcal{T}}]) = tar_{\circ}(\mathring{t})[\sigma_{\mathcal{T}}]$.

**Definition 6.6 (Transition System Specification with Successors)** Let $(\Sigma,\mathcal{R},\mathrm{N})$ be a TSS. A $(\Sigma,\mathcal{R})$-*(successor) literal* is an expression $\mathring{t} \rightsquigarrow_{\mathring{u}} \mathring{v}$ with $\mathring{t},\mathring{u},\mathring{v} \in \mathbb{T}^r(\Sigma,\mathcal{R})$, $src_{\circ}(\mathring{t}) = src_{\circ}(\mathring{u})$ and $src_{\circ}(\mathring{v}) = tar_{\circ}(\mathring{u})$. A *successor rule* over $(\Sigma,\mathcal{R})$ is an expression of the form $\frac{H}{\lambda}$ with $H$ a finite list of $(\Sigma,\mathcal{R})$-literals (the *premises* of the successor rule) and $\lambda$ a $(\Sigma,\mathcal{R})$-literal (the *conclusion*). A *transition system specification with successors (TSSS)* is a tuple $(\Sigma,\mathcal{R},\mathrm{N},\mathcal{U})$ with $(\Sigma,\mathcal{R},\mathrm{N})$ a TSS and $\mathcal{U}$ a set of successor rules over $(\Sigma,\mathcal{R})$.

**Definition 6.7 (Associated LTSS)** For a TSSS $(\Sigma,\mathcal{R},\mathrm{N},\mathcal{U})$, the *associated LTSS* is the LTSS $(S,Tr,source,target,\ell,\rightsquigarrow)$ with $S := \mathrm{P}^r(\Sigma)$, $Tr$ the collection of proofs $\pi$ of closed $\Sigma$-literals $p \xrightarrow{a} q$ from $\mathcal{R}$, where $source(\pi) = p$, $\ell(\pi) = a$ and $target(\pi) = q$, and

$$\rightsquigarrow := \{(t,u,v) \mid \text{a proof of closed } (\Sigma,\mathcal{R})\text{-literal } t \rightsquigarrow_u v \text{ from } \mathcal{U} \text{ exists}\}.$$

# 7 De Simone Languages with Successors

We have enriched standard definitions such as transitions systems and specifications with successors. This allows up to add successors to the De Simone format to define a new congruence format.

**Definition 7.1 (De Simone Format)** A TSSS $(\Sigma,\mathcal{R},\mathrm{N},\mathcal{U})$ is in *De Simone format* if $(\Sigma,\mathcal{R},\mathrm{N})$ is in De Simone format, for every recursive call $\langle X|S\rangle$ and $xa,ya,za \in \mathscr{L}$ it has a successor rule

$$\frac{(tx :: S_X \xrightarrow{xa} x') \rightsquigarrow_{(ty::S_X \xrightarrow{ya} y')} (tz :: y' \xrightarrow{za} z')}{rec_{\chi}(X,S,tx :: S_X \xrightarrow{xa} x') \rightsquigarrow_{rec_{Act}(X,S,ty::S_X \xrightarrow{ya} y')} \mathring{u}}$$

where $\mathring{u} = (tz :: y' \xrightarrow{za} z')$ if $ya \in Act$ and $\mathring{u} = rec_{\chi}(X,S,tx :: S_X \xrightarrow{xa} x')$ otherwise, $rec_{\chi} = rec_{Act}$ if $xa \in Act$ and $rec_{\chi} = rec_{In}$ otherwise, $x',y',z'$ are pairwise distinct process variables not occurring in $\langle X|S\rangle$, and $tx,ty,tz$ are pairwise distinct transition variables. Moreover, each of its other successor rules has the form

$$\frac{\{(tx_i :: x_i \xrightarrow{xa_i} x_i') \rightsquigarrow_{(ty_i::x_i \xrightarrow{ya_i} y_i')} (tz_i :: y_i' \xrightarrow{za_i} z_i') \mid i \in I\}}{\mathrm{R}(xe_1,\ldots,xe_n) \rightsquigarrow_{\mathrm{S}(ye_1,\ldots,ye_n)} \mathring{v}}$$

such that

- $I \subseteq \{1,\ldots,n\}$,
- $x_i,x_i',y_i',z_i'$ for all relevant $i$ are pairwise distinct process variables,
- $tx_i,ty_i,tz_i$ for all relevant $i$ are pairwise distinct transition variables,

- if $i \in I$ then $xe_i = (tx_i :: x_i \xrightarrow{xa_i} x'_i)$ and $ye_i = (ty_i :: x_i \xrightarrow{ya_i} y'_i)$,
- if $i \notin I$ then $xe_i$ is either $x_i$ or $(tx_i :: x_i \xrightarrow{xa_i} x'_i)$, and $ye_i$ is either $x_i$ or $(ty_i :: x_i \xrightarrow{ya_i} y'_i)$,
- $\mathtt{R}$ and $\mathtt{S}$ are $n$-ary transition constructors such that the open transitions $\mathtt{R}(xe_1,\ldots,xe_n)$, $\mathtt{S}(ye_1,\ldots,ye_n)$ and $\mathring{v}$ satisfy
$$src_\circ(\mathtt{R}(xe_1,\ldots,xe_n)) = src_\circ(\mathtt{S}(ye_1,\ldots,ye_n))$$
and $src_\circ(\mathring{v}) = tar_\circ(\mathtt{S}(ye_1,\ldots,ye_n))$,
- $\mathring{v}$ is univariate and contains no other variable expressions than
  - $x_i$ or $(tz_i :: x_i \xrightarrow{za_i} z'_i)$ $(1 \le i \le n \wedge xe_i = ye_i = x_i)$,
  - $(tx_i :: x_i \xrightarrow{xa_i} x'_i)$ $(1 \le i \le n \wedge xe_i \neq x_i \wedge ye_i = x_i)$,
  - $y'_i$ or $(tz_i :: y'_i \xrightarrow{za_i} z'_i)$ $(1 \le i \le n \wedge i \notin I \wedge ye_i \neq x_i)$,
  - $(tz_i :: y'_i \xrightarrow{za_i} z'_i)$ $(i \in I)$, and
- if $\ell_\circ(\mathtt{S}(ye_1,\ldots,ye_n)) \in \mathscr{L}\backslash Act$ then for $i \in I$, $ya_i \in \mathscr{L}\backslash Act$; for $i \notin I$, either $xe_i = x_i$ or $ye_i = x_i$; and $\mathring{v} = \mathtt{R}(ze_1,\ldots,ze_n)$, where
$$ze_i := \begin{cases} (tz_i :: y'_i \xrightarrow{za_i} z'_i) & \text{if } i \in I \\ xe_i & \text{if } i \notin I \text{ and } ye_i = x_i \\ y'_i & \text{otherwise.} \end{cases}$$

The last clause above is simply to ensure that if $t \rightsquigarrow_u v$ for an indicator transition $u$, that is, with $\ell(u) \notin Act$, then $v = t$.

The other conditions of Definition 7.1 are illustrated by the Venn diagram of Figure 1. The outer circle depicts the indices $1,\ldots,n$ numbering the arguments of the operator $Op$ that is the common type of the De Simone rules named $\mathtt{R}$ and $\mathtt{S}$; $I_\mathtt{R}$ and $I_\mathtt{S}$ are the trigger sets of $\mathtt{R}$ and $\mathtt{S}$, respectively. In line with Definition 6.3, $xe = x_i$ for $i \in I_\mathtt{R}$, and $xe = (tx_i :: x_i \xrightarrow{xa_i} x'_i)$ for $i \notin I_\mathtt{R}$. Likewise, $ye = x_i$ for $i \in I_\mathtt{S}$, and $ye = (ty_i :: x_i \xrightarrow{xa_i} y'_i)$ for $i \notin I_\mathtt{S}$. So the premises of any rule named $\mathtt{S}$ are $\{x_i \xrightarrow{xa_i} y'_i \mid i \in I_\mathtt{S}\}$. By Definition 5.1 the target of such a rule is a univariate process expression $q$ with no other variables than $z_1,\ldots,z_n$, where $z_i := x_i$ for $i \in I_\mathtt{S}$ and $z_i := y'_i$ for $i \notin I_\mathtt{S}$. Since $src_\circ(\mathring{v}) = q$, the transition expression $\mathring{v}$ must be univariate, and have no variables other than $ze_i$ for $i = 1,\ldots,n$, where $ze_i$ is either the process variable $z_i$ or a transition variable expression $(tz_i :: z_i \xrightarrow{xa_i} z'_i)$.

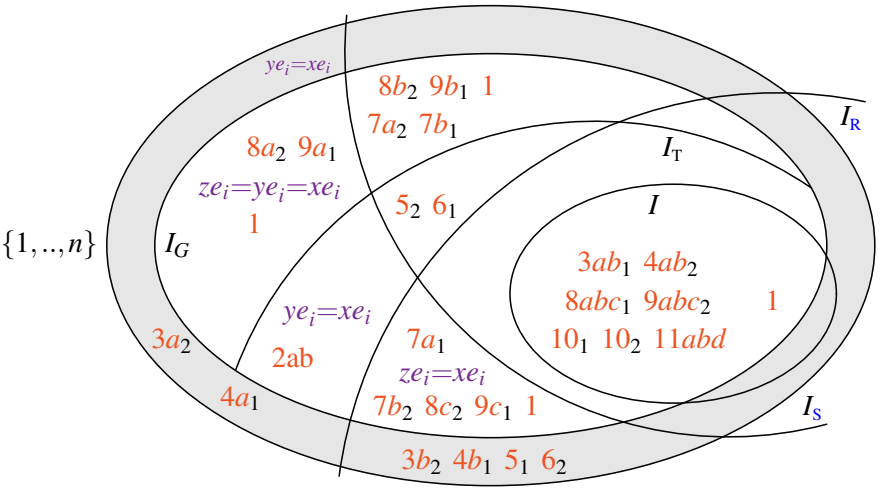



Figure 1: Inclusion between index sets $I, I_\mathtt{R}, I_\mathtt{S}, I_\mathtt{T}, I_G \subseteq \{1,..,n\}$. One has $(I_\mathtt{R} \cap I_G)\backslash I_\mathtt{S} \subseteq I_\mathtt{T}$. The annotations $n_i$ show the location of index $i$ (suppressed for unary operators) of rule $n$.

$I$ is the set of indices $i$ for which the above successor rule has a premise. Since this premise involves the transition variables $tx_i$ and $ty_i$, necessarily $I \subseteq I_{\mathrm{R}} \cap I_{\mathrm{S}}$. Let $I_G$ be the set of indices for which $ze_i$ occurs in $\mathring{v}$, and $I_{\mathrm{T}} \subseteq I_G$ be the subset where $ze_i$ is a transition variable. The conditions on $\mathring{v}$ in Definition 7.1 say that $I \cap I_G \subseteq I_{\mathrm{T}}$ and $(I_{\mathrm{R}} \cap I_G) \setminus I_{\mathrm{S}} \subseteq I_{\mathrm{T}}$. For $i \in I \cap I_G$, the transition variable $tz_i$ is inherited from the premises of the rule, and for $i \in (I_{\mathrm{R}} \cap I_G) \setminus I_{\mathrm{S}}$ the transition variable $tz_i$ is inherited from its source.

In order to show that most classes of indices allowed by our format are indeed populated, we indicated the positions of the indices of the rules of CCS and (the forthcoming) ABCdE from Tables 2 and 5.

Any De Simone language, including CCS, SCCS, ACP and MEIJE, can trivially be extended to a language with successors, e.g. by setting $\mathcal{U} = \emptyset$. This would formalise the assumption that the parallel composition operator of these languages is governed by a *scheduler*, scheduling actions from different components in a nondeterministic way. The choice of $\mathcal{U}$ from Table 2 instead formalises the assumption that parallel components act independently, up to synchronisations between them.

We now present the main theorem of this paper, namely that ep-bisimulation is a lean congruence for all languages that can be presented in De Simone format with successors. A lean congruence preserves equivalence when replacing closed subexpressions of a process by equivalent alternatives. Being a lean congruence implies being a congruence for all operators of the language, but also covers the recursion construct.

**Theorem 7.2 (Lean Congruence)** Ep-bisimulation is a lean congruence for all De Simone languages with successors. Formally, fix a TSSS $(\Sigma, \mathcal{R}, \mathrm{N}, \mathcal{U})$ in De Simone format. If $p \in \mathbb{P}^r(\Sigma)$ and $\rho, \nu$ are two closed $\Sigma$-substitutions with $\forall x \in \mathcal{V}_{\mathcal{P}}.\ \rho(x) \leftrightarrows_{ep} \nu(x)$ then $p[\rho] \leftrightarrows_{ep} p[\nu]$.

The proof can be found in Appendix A of the full version of this paper [15].

In contrast to a lean congruence, a full congruence would also allow replacement within a recursive specification of subexpressions that may contain recursion variables bound outside of these subexpressions. As our proof is already sophisticated, we consider the proof of full congruence to be beyond the scope of the paper. In fact we are only aware of two papers that provide a proof of full congruence via a rule format [21, 9].

We carefully designed our De Simone format with successors and can state the following conjecture.

**Conjecture 7.3** Ep-bisimulation is a full congruence for all De Simone languages with successors.

# 8 A Larger Case Study: The Process Algebra ABCdE

The *Algebra of Broadcast Communication with discards and Emissions* (ABCdE) stems from [13]. It combines CCS [17], its extension with broadcast communication [20, 11, 10], and its extension with signals [4, 6, 7, 10]. Here, we extend CCS as presented in Section 3.

ABCdE is parametrised with sets $\mathscr{C}$ of *handshake communication names* as used in CCS, $\mathscr{B}$ of *broadcast communication names* and $\mathscr{S}$ of *signals*. $\bar{\mathscr{S}} := \{\bar{s} \mid s \in \mathscr{S}\}$ is the set of signal emissions. The collections $\mathscr{B}!$, $\mathscr{B}?$ and $\mathscr{B}:$ of *broadcast*, *receive*, and *discard* actions are given by $\mathscr{B}\sharp := \{b\sharp \mid b \in \mathscr{B}\}$ for $\sharp \in \{!, ?, :\}$. $Act := \mathscr{C} \cup \bar{\mathscr{C}} \cup \{\tau\} \cup \mathscr{B}! \cup \mathscr{B}? \cup \mathscr{S}$ is the set of *actions*, with $\tau$ the *internal action*, and $\mathscr{L} := Act \cup \mathscr{B}: \cup \bar{\mathscr{S}}$ is the set of *transition labels*. Complementation extends to $\mathscr{C} \cup \bar{\mathscr{C}} \cup \mathscr{S} \cup \bar{\mathscr{S}}$ by $\bar{\bar{c}} := c$.

Below, $c$ ranges over $\mathscr{C} \cup \bar{\mathscr{C}} \cup \mathscr{S} \cup \bar{\mathscr{S}}$, $\eta$ over $\mathscr{C} \cup \bar{\mathscr{C}} \cup \{\tau\} \cup \mathscr{S} \cup \bar{\mathscr{S}}$, $\alpha$ over $Act$, $\ell$ over $\mathscr{L}$, $\gamma$ over $In := \mathscr{L} \setminus Act$, $b$ over $\mathscr{B}$, $\sharp, \sharp_1, \sharp_2$ over $\{!, ?, :\}$, $s$ over $\mathscr{S}$, $S$ over recursive specifications and $X$ over $V_S$. A *relabelling* is a function $f : (\mathscr{C} \to \mathscr{C}) \cup (\mathscr{B} \to \mathscr{B}) \cup (\mathscr{S} \to \mathscr{S})$; it extends to $\mathscr{L}$ by $f(\bar{c}) = \overline{f(c)}$, $f(\tau) := \tau$ and $f(b\sharp) = f(b)\sharp$.

Next to the constant and operators of CCS, the process signature $\Sigma$ of ABCdE features a unary *signalling* operator $\_\hat{}s$ for each signal $s \in \mathscr{S}$.

Table 3: Structural operational semantics of ABCdE

$$\frac{}{\mathbf{0} \xrightarrow{b:} \mathbf{0}}\ b{:}\mathbf{0} \qquad \frac{\alpha \neq b?}{\alpha.x \xrightarrow{b:} \alpha.x}\ b{:}\alpha. \qquad \frac{x \xrightarrow{b:} x',\ y \xrightarrow{b:} y'}{x+y \xrightarrow{b:} x'+y'}\ +_{\mathrm{C}}$$

$$\frac{x \xrightarrow{b\sharp_1} x',\ y \xrightarrow{b\sharp_2} y' \quad (\sharp_1 \circ \sharp_2 = \sharp \neq \_)}{x|y \xrightarrow{b\sharp} x'|y'}\ |_{\mathrm{C}} \qquad \text{with}$$

| $\circ$ | $!$ | $?$ | $:$ |
|---|---|---|---|
| $!$ | $\_$ | $!$ | $!$ |
| $?$ | $!$ | $?$ | $?$ |
| $:$ | $!$ | $?$ | $:$ |

$$\frac{}{x\hat{}s \xrightarrow{\bar{s}} x\hat{}s}\ (\rightarrow^{s}) \qquad \frac{x \xrightarrow{\bar{s}} x'}{x+y \xrightarrow{\bar{s}} x'+y}\ +^{e}_{L} \qquad \frac{y \xrightarrow{\bar{s}} y'}{x+y \xrightarrow{\bar{s}} x+y'}\ +^{e}_{\mathrm{R}}$$

$$\frac{x \xrightarrow{\alpha} x'}{x\hat{}s \xrightarrow{\alpha} x'}\ \hat{}s_{Act} \qquad \frac{x \xrightarrow{\gamma} x'}{x\hat{}s \xrightarrow{\gamma} x'\hat{}s}\ \hat{}s_{In} \qquad \frac{\langle S_X|S\rangle \xrightarrow{\gamma} y}{\langle X|S\rangle \xrightarrow{\gamma} \langle X|S\rangle}\ rec_{In}$$

The semantics of ABCdE is given by the transition rule templates displayed in Tables 1 and 3. The latter augments CCS with mechanisms for broadcast communication and signalling. The rule $|_{\mathrm{C}}$ presents the core of broadcast communication [20], where any broadcast-action $b!$ performed by a component in a parallel composition needs to synchronise with either a receive action $b?$ or a discard action $b:$ of any other component. In order to ensure associativity of the parallel composition, rule $|_{\mathrm{C}}$ also allows receipt actions of both components ($\sharp_1 = \sharp_2 = ?$), or a receipt and a discard, to be combined into a receipt action.

A transition $p \xrightarrow{b:} q$ is derivable only if $q = p$. It indicates that the process $p$ is unable to receive a broadcast communication $b!$ on channel $b$. The Rule $b{:}\mathbf{0}$ allows the nil process (inaction) to discard any incoming message; in the same spirit $b{:}\alpha.$ allows a message to be discarded by a process that cannot receive it. A process offering a choice can only perform a discard-action if both choice-options can discard the corresponding broadcast (Rule $+_{\mathrm{C}}$). Finally, by rule $rec_{In}$, a recursively defined process $\langle X|S\rangle$ can discard a broadcast iff $\langle S_X|S\rangle$ can discard it. The variant $rec_{In}$ of $rec_{Act}$ is introduced to maintain the property that $target(\theta) = source(\theta)$ for any indicator transition $\theta$.

A signalling process $p\hat{}s$ emits the signal $s$ to be read by another process. A typically example is a traffic light being red. Signal emission is modelled as an indicator transition, which does not change the state of the emitting process. The first rule $(\rightarrow^{s})$ models the emission $\bar{s}$ of signal $s$ to the environment. The environment (processes running in parallel) can read the signal by performing a read action $s$. This action synchronises with the emission $\bar{s}$, via rule $|_{\mathrm{C}}$ of Table 1. Reading a signal does not change the state of the emitter.

Rules $+^{e}_{L}$ and $+^{e}_{\mathrm{R}}$ describe the interaction between signal emission and choice. Relabelling and restriction are handled by rules $\backslash L$ and $[f]$ of Table 1, respectively. These operators do not prevent the emission of a signal, and emitting signals never changes the state of the emitting process. Signal emission $p\hat{}s$ does not block other transitions of $p$.

It is trivial to check that the TSS of ABCdE is in De Simone format.

The transition signature of ABCdE (Table 4) is completely determined by the set of transition rule templates in Tables 1 and 3. We have united the rules for handshaking and broadcast communication by assigning the same name $|_{\mathrm{C}}$ to all their instances. When expressing transitions in ABCdE as expressions, we use infix notation for the binary transition constructors, and prefix or postfix notation for unary ones. For example, the transition $b{:}\mathbf{0}()$ is shortened to $b{:}\mathbf{0}$, $\xrightarrow{\alpha}(p)$ to $\xrightarrow{\alpha}p$, $\backslash L(t)$ to $t\backslash L$, and $|_{\mathrm{L}}(t,p)$ to $t|_{\mathrm{L}}p$.

Table 4: Transition signature of ABCdE

| Constructor | $\xrightarrow{\alpha}$ | $(\rightarrow^s)$ | $b{:}\mathbf{0}$ | $b{:}\alpha.$ | $+_{\mathrm{L}}$ | $+_{\mathrm{R}}$ | $+_{\mathrm{C}}$ | $+^{e}_{L}$ | $+^{e}_{\mathrm{R}}$ | $\mid_{\mathrm{L}}$ | $\mid_{\mathrm{C}}$ | $\mid_{\mathrm{R}}$ | $\backslash L$ | $[f]$ | $\hat{}s_{Act}$ | $\hat{}s_{In}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Arity | 1 | 1 | 0 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 |
| Trigger Set | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\{1\}$ | $\{2\}$ | $\{1,2\}$ | $\{1\}$ | $\{2\}$ | $\{1\}$ | $\{1,2\}$ | $\{2\}$ | $\{1\}$ | $\{1\}$ | $\{1\}$ | $\{1\}$ |

Table 5: Successor rules for ABCdE

$$\frac{\ell(t)\in\{b?,b{:}\}}{\xrightarrow{b?}P\rightsquigarrow_{\xrightarrow{b?}P}t}\ 2a^b \qquad \frac{\ell(\zeta)\in\mathscr{B}{:}\cup\bar{\mathscr{S}}}{\chi\rightsquigarrow_{\zeta}\chi}\ 1 \qquad \frac{\ell(t)\in\{b?,b{:}\}}{b{:}\alpha.P\rightsquigarrow_{\xrightarrow{\alpha}P}t}\ 2b^b$$

$$\frac{t\rightsquigarrow_v t'}{t+^{e}_{L}Q\rightsquigarrow_{v+_{\mathrm{L}}Q}t'}\ 3a \qquad \frac{t\rightsquigarrow_v t'}{t+_{\mathrm{C}}u\rightsquigarrow_{v+_{\mathrm{L}}Q}t'}\ 3b \qquad \frac{u\rightsquigarrow_w u'}{P+^{e}_{\mathrm{R}}u\rightsquigarrow_{P+_{\mathrm{R}}w}u'}\ 4a \qquad \frac{u\rightsquigarrow_w u'}{t+_{\mathrm{C}}u\rightsquigarrow_{P+_{\mathrm{R}}w}u'}\ 4b$$

$$\frac{\ell(t)=b?\quad \ell(u')\in\{b?,b{:}\}}{\begin{array}{l}t+_{\mathrm{L}}Q\rightsquigarrow_{P+_{\mathrm{R}}w}u'\\ t+^{e}_{L}Q\rightsquigarrow_{P+_{\mathrm{R}}w}u'\end{array}}\ 5^b \qquad \frac{\ell(u)=b?\quad \ell(t')\in\{b?,b{:}\}}{\begin{array}{l}P+_{\mathrm{R}}u\rightsquigarrow_{v+_{\mathrm{L}}Q}t'\\ P+^{e}_{\mathrm{R}}u\rightsquigarrow_{v+_{\mathrm{L}}Q}t'\end{array}}\ 6^b$$

$$\frac{t\rightsquigarrow_v t'}{rec_{In}(X,S,t)\rightsquigarrow_{rec_{Act}(X,S,v)}t'}\ 11c \qquad \frac{t\rightsquigarrow_v t'}{\begin{array}{l}t\hat{}s_{Act}\rightsquigarrow_{\hat{v}s_{Act}}t'\\ t\hat{}s_{In}\rightsquigarrow_{\hat{v}s_{Act}}t'\end{array}}\ 11d$$

| Meta | Variable Expression |
|---|---|
| $P$ | $x_1$ |
| $Q$ | $x_2$ |
| $P'$ | $y'_1$ |
| $Q'$ | $y'_2$ |
| $t$ | $(tx_1 :: x_1 \xrightarrow{xa_1} x'_1)$ |
| $u$ | $(tx_2 :: x_2 \xrightarrow{xa_2} x'_2)$ |
| $v$ | $(ty_1 :: x_1 \xrightarrow{ya_1} y'_1)$ |
| $w$ | $(ty_2 :: x_2 \xrightarrow{ya_2} y'_2)$ |
| $t'$ | $(tz_1 :: y'_1 \xrightarrow{za_1} z'_1)$ |
| $u'$ | $(tz_2 :: y'_2 \xrightarrow{za_2} z'_2)$ |

Table 5 extends the successor relation of CCS (Table 2) to ABCdE. $P,Q$ are process variables, $t,v$ transition variables enabled at $P$, $u,w$ transition variables enabled at $Q$, $P',Q'$ the targets of $v,w$, respectively and $t',u'$ transitions enabled at $P',Q'$, respectively. To express those rules in the same way as Definition 7.1, we replace the metavariables $P$, $Q$, $t$, $u$, etc. with variable expressions as indicated on the right. Here $xa_i$, $ya_i$, $za_i$ are label variables that should be instantiated to match the trigger of the rules and side conditions. As ABCdE does not feature operators of arity $>2$, the index $i$ from Definition 7.1 can be 1 or 2 only.

To save duplication of rules 8b, 8c, 9b, 9c and 10 we have assigned the same name $\mid_{\mathrm{C}}$ to the rules for handshaking and broadcast communication. The intuition of the rules of Table 5 is explained in detail in [13].

In the naming convention for transitions from [13] the sub- and superscripts of the transition constructors $+$, $\mid$ and $\hat{}s$, and of the recursion construct, were suppressed. In most cases that yields no ambiguity, as the difference between $\mid_{\mathrm{L}}$ and $\mid_{\mathrm{R}}$, for instance, can be detected by checking which of its two arguments are of type transition versus process. Moreover, it avoids the duplication in rules 3a, 4a, 5, 6, 11c and 11d. The ambiguity between $+_{\mathrm{L}}$ and $+^{e}_{L}$ (or $+_{\mathrm{R}}$ and $+^{e}_{\mathrm{R}}$) was in [13] resolved by adorning rules 3–6 with a side condition $\ell(v)\notin\bar{\mathscr{S}}$ or $\ell(w)\notin\bar{\mathscr{S}}$, and the ambiguity between $rec_{Act}$ and $rec_{In}$ (or $\hat{}s_{Act}$ and $\hat{}s_{In}$) by adorning rules 11c and 11d with a side condition $\ell(v)\in Act$; this is not needed here.

It is easy to check that all rules are in the newly introduced De Simone format, except Rule 1. However, this rule can be converted in to a collection of De Simone rules by substituting $\mathrm{R}(xe_1,\ldots,xe_n)$ for $\chi$ and $\mathrm{S}(ye_1,\ldots,ye_n)$ for $\zeta$, adding a premise in the form of $xe_i\rightsquigarrow_{ye_i}(tz_i :: y'_i\xrightarrow{za_i}z'_i))$ if $i\in I_{\mathrm{R}}\cap I_{\mathrm{S}}$,

for each pair of rules of the same type named R and S.[3] The various occurrences of 1 in Figure 1 refer to these substitution instances. It follows that $\leftrightarrows_{ep}$ is a congruence for the operators of ABCdE, as well as a lean congruence for recursion.

# 9 Related Work & Conclusion

In this paper we have added a successor relation to the well-known De Simone format. This has allowed us to prove the general result that enabling preserving bisimilarity – a finer semantic equivalence relation than strong bisimulation – is a lean congruence for all languages with a structural operational semantics within this format. We do not cover full congruence yet, as proofs for general recursions are incredible hard and usually excluded from work justifying semantic equivalences.

There is ample work on congruence formats in the literature. Good overview papers are [1, 19]. For system description languages that do not capture time, probability or other useful extensions to standard process algebras, all congruence formats target strong bisimilarity, or some semantic equivalence or preorder that is strictly coarser than strong bisimilarity. As far as we know, the present paper is the first to define a congruence format for a semantic equivalence that is finer than strong bisimilarity.

Our congruence format also ensures a lean congruence for recursion. So far, the only papers that provide a rule format yielding a congruence property for recursion are [21] and [9], and both of them target strong bisimilarity.

In Sections 3 and 8, we have applied our format to show lean congruence of ep-bisimilarity for the process algebra CCS and ABCdE, respectively. This latter process algebra features broadcast communication and signalling. These two features are representative for issues that may arise elsewhere, and help to ensure that our results are as general as possible. Our congruence format can effortlessly be applied to other calculi like CSP [5] or ACP [3].

In order to evaluate ep-bisimilarity on process algebras like CCS, CSP, ACP or ABCdE, their semantics needs to be given in terms of labelled transition systems extended with a successor relation $\rightsquigarrow$. This relation models concurrency between transitions enabled in the same state, and also tells what happens to a transition if a concurrent transition is executed first. Without this extra component, labelled transition systems lack the necessary information to capture liveness properties in the sense explained in the introduction.

In a previous paper [13] we already gave such a semantics to ABCdE. The rules for the successor relation presented in [13], displayed in Tables 2 and 5, are now seen to fit our congruence format. We can now also conclude that ep-bisimulation is a lean congruence for ABCdE. In [14, Appendix B] we contemplate a very different approach for defining the relation $\rightsquigarrow$. Following [10], we understand each transition as the synchronisation of a number of elementary particles called *synchrons*. Then relations on synchrons are proposed in terms of which the $\rightsquigarrow$-relation is defined. It is shown that this leads to the same $\rightsquigarrow$-relation as the operational approach from [13] and Tables 2 and 5.

[3] This yields $1^2+2\cdot1+5\cdot3+3\cdot2+2\cdot1=26$ rules of types $(\mathbf{0},0)$, $(\alpha._,1)$, $(+,2)$, $(\hat{}s,1)$ and $\langle X|S\rangle$ not included in Tables 2 and 5.

## References


[1] L. Aceto, W. Fokkink & C. Verhoef (2000): *Structural Operational Semantics*. In J. Bergstra, A. Ponse & S. Smolka, editors: *Handbook of Process Algebra*, chapter 3, Springer, pp. 197–292.

[2] D. Austry & G. Boudol (1984): *Algèbre de Processus et Synchronisation*. *Theoretical Computer Science* 30, pp. 91–131, doi:10.1016/0304-3975(84)90067-7.

[3] J.C.M. Baeten & W.P. Weijland (1990): *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, doi:10.1017/CBO9780511624193.

[4] J.A. Bergstra (1988): *ACP with signals*. In J. Grabowski, P. Lescanne & W. Wechler, editors: Proc. International Workshop on *Algebraic and Logic Programming*, LNCS 343, Springer, pp. 11–20, doi:10.1007/3-540-50667-5_53.

[5] S.D. Brookes, C.A.R. Hoare & A.W. Roscoe (1984): *A theory of communicating sequential processes*. *Journal of the ACM* 31(3), pp. 560–599, doi:10.1145/828.833.

[6] F. Corradini, M.R. Di Berardini & W. Vogler (2009): *Time and Fairness in a Process Algebra with Non-blocking Reading*. In M. Nielsen, A. Kucera, P.B. Miltersen, C. Palamidessi, P. Tuma & F.D. Valencia, editors: *Theory and Practice of Computer Science*, SOFSEM'09, LNCS 5404, Springer, pp. 193–204, doi:10.1007/978-3-540-95891-8_20.

[7] V. Dyseryn, R.J. van Glabbeek & P. Höfner (2017): *Analysing Mutual Exclusion using Process Algebra with Signals*. In K. Peters & S. Tini, editors: Proceedings Combined 24th International Workshop on *Expressiveness in Concurrency* and 14th Workshop on *Structural Operational Semantics*, Berlin, Germany, 4th September 2017, *Electronic Proceedings in Theoretical Computer Science* 255, Open Publishing Association, pp. 18–34, doi:10.4204/EPTCS.255.2.

[8] R.J. van Glabbeek (1994): *On the expressiveness of ACP (extended abstract)*. In A. Ponse, C. Verhoef & S.F.M. van Vlijmen, editors: Proceedings First Workshop on the *Algebra of Communicating Processes*, ACP'94, Utrecht, The Netherlands, May 1994, Workshops in Computing, Springer, pp. 188–217, doi:10.1007/978-1-4471-2120-6_8.

[9] R.J. van Glabbeek (2017): *Lean and Full Congruence Formats for Recursion*. In: Proceedings $32^{nd}$ Annual ACM/IEEE Symposium on *Logic in Computer Science*, LICS 2017, IEEE Computer Society Press, doi:10.1109/LICS.2017.8005142.

[10] R.J. van Glabbeek (2019): *Justness: A Completeness Criterion for Capturing Liveness Properties (extended abstract)*. In M. Bojańczyk & A. Simpson, editors: Proceedings 22st International Conference on *Foundations of Software Science and Computation Structures* (FoSSaCS'19); held as part of the *European Joint Conferences on Theory and Practice of Software* (ETAPS'19), Prague, Czech Republic, April 2019, LNCS 11425, Springer, pp. 505–522, doi:10.1007/978-3-030-17127-8_29.

[11] R.J. van Glabbeek & P. Höfner (2015): *Progress, Fairness and Justness in Process Algebra*. Technical Report 8501, NICTA, Sydney, Australia. Available at `http://arxiv.org/abs/1501.03268`.

[12] R.J. van Glabbeek & P. Höfner (2019): *Progress, Justness and Fairness*. *ACM Computing Surveys* 52(4):69, doi:10.1145/3329125.

[13] R.J. van Glabbeek, P. Höfner & W. Wang (2021): *Enabling Preserving Bisimulation Equivalence*. In S. Haddad & D. Varacca, editors: Proceedings 32nd International Conference on *Concurrency Theory*, CONCUR'21, *Leibniz International Proceedings in Informatics (LIPIcs)* 203, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, doi:10.4230/LIPIcs.CONCUR.2021.33.

[14] R.J. van Glabbeek, P. Höfner & W. Wang (2021): *Enabling Preserving Bisimulation Equivalence*. Available at `https://arxiv.org/abs/2108.00142`. Full version of [13].

[15] R.J. van Glabbeek, P. Höfner & W. Wang (2023): *A Lean-Congruence Format for EP-Bisimilarity*. arXiv:2308.16350. Full version of this paper.

[16] J.F. Groote & F.W. Vaandrager (1992): *Structured Operational Semantics and Bisimulation as a Congruence*. *Information and Computation* 100(2), pp. 202–260, doi:10.1016/0890-5401(92)90013-6.

[17] R. Milner (1990): *Operational and algebraic semantics of concurrent processes*. In J. van Leeuwen, editor: *Handbook of Theoretical Computer Science*, chapter 19, Elsevier Science Publishers B.V. (North-Holland), pp. 1201–1242. Alternatively see *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, 1989, of which an earlier version appeared as *A Calculus of Communicating Systems*, LNCS 92, Springer, 1980, doi:10.1007/3-540-10235-3.

[18] R. Milner (1983): *Calculi for Synchrony and Asynchrony*. *Theoretical Computer Science* 25, pp. 267–310, doi:10.1016/0304-3975(83)90114-7.

[19] M.R. Mousavi, M.A. Reniers & J.F. Groote (2007): *SOS formats and meta-theory: 20 years after*. *Theoretical Computer Science* 373(3), pp. 238–272, doi:10.1016/j.tcs.2006.12.019.

[20] K.V.S. Prasad (1991): *A Calculus of Broadcasting Systems*. In S. Abramsky & T.S.E. Maibaum, editors: Proceedings of the International Joint Conference on *Theory and Practice of Software Development*, TAPSOFT'91, *Volume 1: Colloquium on Trees in Algebra and Programming*, CAAP'91, LNCS 493, Springer, pp. 338–358, doi:10.1007/3-540-53982-4_19.

[21] A. Rensink (2000): *Bisimilarity of Open Terms*. *Information and Computation* 156(1-2), pp. 345–385, doi:10.1006/inco.1999.2818.

[22] R. de Simone (1985): *Higher-level synchronising devices in* MEIJE-*SCCS*. *Theoretical Computer Science* 37, pp. 245–267, doi:10.1016/0304-3975(85)90093-3.

# Using $\pi$-Calculus Names as Locks

Daniel Hirschkoff
ENS de Lyon
daniel.hirschkoff@ens-lyon.fr

Enguerrand Prebet
Karlsruhe Institute of Technology
enguerrand.prebet@kit.edu

Locks are a classic data structure for concurrent programming. We introduce a type system to ensure that names of the asynchronous $\pi$-calculus are used as locks. Our calculus also features a construct to deallocate a lock once we know that it will never be acquired again. Typability guarantees two properties: deadlock-freedom, that is, no acquire operation on a lock waits forever; and leak-freedom, that is, all locks are eventually deallocated.

We leverage the simplicity of our typing discipline to study the induced typed behavioural equivalence. After defining barbed equivalence, we introduce a sound labelled bisimulation, which makes it possible to establish equivalence between programs that manipulate and deallocate locks.

## 1 Introduction

The $\pi$-calculus is an expressive process calculus based on the notion of name, in which name-passing is the primitive notion of interaction between processes. Processes of the $\pi$-calculus have been used to represent several aspects of programming, like data structures, protocols, or constructs such as functions, continuations, objects, and references. The $\pi$-calculus also comes with a well-developed theory of behavioural equivalence. This theory can be exploited to reason about contextual equivalence in programming languages, by translating programs as $\pi$-calculus processes.

In this work, we follow this path for locks, a basic data structure for concurrent programming. We study how $\pi$-calculus names can be used to represent locks. We show that the corresponding programming discipline in the $\pi$-calculus induces a notion of behavioural equivalence between processes, which can be used to reason about processes manipulating locks. This approach has been followed to analyse several disciplines for the usage of $\pi$-calculus names: linearity [15], receptiveness [25], locality [16], internal mobility [24], functions [23, 5], references [7, 21].

It is natural to represent locks in $A\pi$, the asynchronous version of the $\pi$-calculus [1, 9]. A lock is referred to using a $\pi$-calculus name. It is represented as an asynchronous output: the release of the lock. Dually, an input represents the acquire operation on some lock.

In this paper, we introduce $\pi\ell$w, a version of the asynchronous $\pi$-calculus with only lock names. Two properties should be ensured for names to be used as locks: first, a lock can appear at most once in released form. Second, acquiring a lock entails the obligation to release it. For instance, process $\ell_1(x).(\overline{\ell_1}\langle x\rangle \mid \overline{\ell_2}\langle x\rangle)$ has these properties: the process acquires lock $\ell_1$, then releases it, together with lock $\ell_2$. We remark that this this process owns lock $\ell_2$, which is released after $\ell_1$ is acquired. We show that a simple type system can be defined to guarantee the two properties mentioned above.

When manipulating locks, it is essential to avoid the program from getting stuck in a state where a lock needs to be acquired but cannot be released. Consider the following process:

$$P_{\mathsf{dl}} \stackrel{\text{def}}{=} \ell_1(x).(\overline{\ell_1}\langle x\rangle \mid \overline{\ell_2}\langle x\rangle) \mid \ell_2(y).(\overline{\ell_1}\langle y\rangle \mid \overline{\ell_2}\langle y\rangle).$$

G. Caltais and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2023).
EPTCS 387, 2023, pp. 76–96, doi:10.4204/EPTCS.387.7

The subprocess on the left needs to acquire lock $\ell_1$, which is owned by the other subprocess, and symmetrically: this is a deadlock. Our type system rules out processes that exhibit this kind of cyclic dependency between locks. This is achieved by controlling parallel composition: two processes in parallel can share at most one lock name. Process $P_{\mathsf{dl}}$ thus cannot be typed, because names $\ell_1$ and $\ell_2$ are shared between the two subprocesses. The acyclicity property enjoyed by typable processes yields deadlock-freedom.

To avoid situations where a lock is in released state and cannot be accessed, $\pi\ell$w also features a construct to deallocate a lock, called *wait*, inspired from [12]. Process $\ell((x)).P$ waits until no acquire is pending on lock $\ell$, at which point it deallocates $\ell$, reading the final value stored in $\ell$ as $x$. The reduction rule for wait is

$$(\nu\ell)\left(\overline{\ell}\langle v\rangle \mid \ell((x)).P\right) \to P\{^{v}/_{x}\} \tag{1}$$

provided $\ell$ is not among the free names of $P$. In the reduction above, the restriction on $\ell$ disappears after the last interaction involving $\ell$ has taken place.

The main contributions of this work are the following:

- We introduce $\pi\ell$w, a $\pi$-calculus with higher-order locks (in the sense that locks can be stored in locks) and lock deallocation. The type system for $\pi\ell$w controls the usage and the sharing of lock names between processes. We provide some examples to illustrate how locks can be manipulated according to the programming discipline induced by types.
- We show that typable processes in $\pi\ell$w enjoy deadlock- and leak-freedom. The proofs rely on simple arguments involving the graph induced by the sharing of locks among processes.
- We analyse typed behavioural equivalence in $\pi\ell$w. Types restrict the set of contexts that can interact with processes, yielding a coarser behavioural equivalence than in the untyped case.

  We first introduce typed barbed equivalence, written $\simeq_w$. Relation $\simeq_w$ is defined by observing the behaviour of processes when they are placed in typable contexts. We then express the interactions between typed processes and typed context by means of a Labelled Transition System (LTS) that takes into account typing constraints. This allows us to introduce *typed bisimilarity*, $\approx_w$, the main proof technique to establish barbed equivalence: we indeed prove soundness, that is, $\approx_w \subseteq \simeq_w$.

  We discuss several examples that help to understand how we can reason about behavioural equivalence in $\pi\ell$w. We are not aware of existing labelled equivalences taking into account name deallocation in the $\pi$-calculus.

Beyond $\pi\ell$w, we believe that $\approx_w$ can be used as a building block when reasoning in the $\pi$-calculus about programs that use various features, among which locks.

The aforementioned contributions are presented in two steps. We first introduce $\pi\ell$, an asynchronous $\pi$-calculus with higher-order locks. $\pi\ell$w is obtained by adding the wait construct to $\pi\ell$. Several important ideas can be presented in $\pi\ell$, and we can build on the notions introduced for $\pi\ell$ to extend them for $\pi\ell$w.

We now highlight some of the technical aspects involved in our work.

The type system for $\pi\ell$ guarantees deadlock-freedom, in the sense that for typable processes, an acquire operation cannot be blocked forever. This holds for *complete processes*: a process is complete if for every lock $\ell$ it uses, a release of $\ell$ is available. Availability need not be immediate, for instance the release operation on lock $\ell$ may be blocked by an acquire on $\ell'$. We prove progress based on the fact that the type system guarantees acyclicity of the dependence relation between locks. Progress entails deadlock-freedom.

When adding the wait construct, we rely on a similar reasoning to prove leak-freedom for $\pi\ell$w, which in our setting means that all locks are eventually deallocated. The type system for $\pi\ell$w is richer than the

one for $\pi\ell$ not only because it takes wait into account, but also because it makes it possible to transmit the obligation of releasing or deallocating a lock via another lock. For instance, it is possible, depending on the type of $\ell$, that in process $\ell(\ell').P$, the continuation $P$ has the obligation not only to release lock $\ell$, but also to deallocate $\ell'$, or release $\ell'$, or both.

To define typed barbed equivalence in $\pi\ell$, written $\simeq$, we must take into account deadlock-freedom, which has several consequences. First, we observe complete processes: intuitively, computations in $\pi\ell$ make sense only for such processes, and a context interacting with a process should not be able to block a computation by never performing some release operation. Second, all barbs are always observable in $\pi\ell$. In other words, if $\ell$ is a free name of a complete typable process $P$, then $P$ can never loose the ability to release $\ell$. This is in contrast with barbed equivalence in the $\pi$-calculus, or in CCS, where the absence of a barb can be used to observe behaviours. We therefore adopt a stronger notion of barb, where the value stored in a lock, and not only the name of the lock, can be observed.

The ideas behind $\simeq$ are used to define $\simeq_w$, typed barbed equivalence in $\pi\ell$w. A challenge when defining typed bisimilarity in $\pi\ell$w is to come up with labelled transitions corresponding to the reduction in (1). Intuitively, if $P \xrightarrow{\ell((v))} P'$ ($P$ deallocates $\ell$ and continues as $P'$), we must make sure that this transition is the last interaction at $\ell$. We define a typed LTS to handle name deallocation, and show that bisimilarity is sound for barbed equivalence in $\pi\ell$w.

**Paper outline.** We study $\pi\ell$ in Section 2. We first expose the essential ideas of our deadlock-freedom proof in CCS$\ell$, a simple version of the Calculus of Communicating Systems [18] with lock names. After extending these results to $\pi\ell$, we define barbed equivalence for $\pi\ell$, written $\simeq$. We provide a labelled semantics that is sound for $\simeq$, and present several examples of behavioural equivalences in $\pi\ell$. In Section 3, we add the wait construct, yielding $\pi\ell$w. We show how to derive leak-freedom, and define a labelled semantics, building on the ideas of Section 2. We discuss related and future work in Section 4.

## 2 $\pi\ell$, a Deadlock-Free Asynchronous $\pi$-Calculus

We present deadlock-freedom in the simple setting of CCS$\ell$ in Section 2.1. This approach is extended to handle higher-order locks in $\pi\ell$ (Section 2.2). We study behavioural equivalence in $\pi\ell$ in Section 2.3.

### 2.1 CCS$\ell$: Ensuring Deadlock-Freedom using Composition

CCS$\ell$ is a simplification of $\pi\ell$, to present the ideas underlying the type system and the proof of deadlock-freedom. CCS$\ell$ is defined as an asynchronous version of CCS with acquire and release operations. We postulate the existence of an infinite set of *lock names*, written $\ell, \ell', \ell_1, \ldots$, which we often simply call names. CCS$\ell$ processes are defined by the following grammar:

$$P ::= \ell.P \mid \overline{\ell} \mid (\nu\ell)P \mid P_1 \mid P_2.$$

$\overline{\ell}$ is the release of lock $\ell$. Process $\ell.P$ acquires $\ell$ and then acts as $P$—we say that $P$ performs an *acquire on* $\ell$. There is no $\mathbf{0}$ process in CCS$\ell$, intuitively because we do not take into consideration processes with no lock at all. Restriction is a binder, and we write $\mathrm{fln}(P)$ for the set of free lock names in $P$. If $\mathbb{S} = \{\ell_1, \ldots, \ell_k\}$ is a set of lock names, we write $(\nu\mathbb{S})P$ for $(\nu\ell_1)\ldots(\nu\ell_k)P$.

The definition of structural congruence, written $\equiv$, and reduction, written $\rightarrow$, are standard. They are given in Appendix A.1. Relation $\Rightarrow$ is the transitive reflexive closure of $\rightarrow$.

**Type System.** To define the type system for CCS$\ell$, we introduce typing environments. We use $\gamma$ to range over sets of lock names. We write $\gamma_1\#\gamma_2$ whenever $\gamma_1 \cap \gamma_2 = \emptyset$. We write $\gamma,\ell$ for the set $\gamma \uplus \{\ell\}$: the notation implicitly imposes $\ell \notin \gamma$.

*Typing environments*, written $\Gamma$, are sets of such sets, with the additional constraint that these should be pairwise disjoint. We write $\Gamma = \gamma_1,\ldots,\gamma_k$, for $k \geq 1$, to mean that $\Gamma$ is equal to $\{\gamma_1,\ldots,\gamma_k\}$, with $\gamma_i\#\gamma_j$ whenever $i \neq j$. The $\gamma_i$s are called the *components* of $\Gamma$ in this case, and $\mathrm{dom}(\Gamma)$, the domain of $\Gamma$, is defined as $\gamma_1 \cup \cdots \cup \gamma_k$. We write $\Gamma_1\#\Gamma_2$ whenever $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$.

As for components $\gamma$, the notation $\Gamma,\gamma$ stands for a set (of sets) that can be written as $\Gamma \uplus \{\gamma\}$. Using these two notations together, we can write $\Gamma,\gamma,\ell$ to refer to a typing environment containing a component that contains $\ell$. We sometimes add parentheses, writing e.g. $\Gamma,(\gamma,\ell,\ell')$, to ease readability.

The typing judgement is of the form $\Gamma;\mathbb{R} \vdash P$, where $\mathbb{R}$ is a set of lock names. If $\Gamma;\mathbb{R} \vdash P$, then $\mathbb{R}$ is the set of locks owned by $P$, that must be released. Moreover any component $\gamma$ of $\Gamma$ intuitively corresponds to a subprocess of $P$ that only accesses the names in $\gamma$. Here, accessing a lock name $\ell$ means either releasing $\ell$ or performing an acquire on $\ell$, or both. The typing rules are as follows:

$$\textsc{Acq-C}\ \frac{\Gamma,(\gamma,\ell);\mathbb{R},\ell \vdash P}{\{\mathsf{flatten}(\Gamma) \uplus (\gamma,\ell)\};\mathbb{R} \vdash \ell.P} \qquad \textsc{Rel-C}\ \frac{}{\Gamma,(\gamma,\ell);\{\ell\} \vdash \overline{\ell}} \qquad \textsc{New-C}\ \frac{\Gamma,(\gamma,\ell);\mathbb{R},\ell \vdash P}{\Gamma,\gamma;\mathbb{R} \vdash (\nu\ell)P} \qquad \textsc{Par-C}\ \frac{\Gamma_1;\mathbb{R}_1 \vdash P_1 \qquad \Gamma_2;\mathbb{R}_2 \vdash P_2}{\Gamma_1 \bullet \Gamma_2;\mathbb{R}_1 \uplus \mathbb{R}_2 \vdash P_1 \mid P_2}$$

In rule Acq-C, operator flatten has the effect of mergining all components in a typing environment into a single component. In particular, if $\Gamma = \{\gamma_1,\ldots,\gamma_k\}$, then $\mathsf{flatten}(\Gamma)$ stands for $\gamma_1 \uplus \cdots \uplus \gamma_k$. Intuitively, the causal dependency introduced by the prefix $\ell.P$ induces a dependence between $\ell$ and all the locks in $P$, forcing these locks to belong to the same component.

In the typing rules, we write $\mathbb{R},\ell$ for $\mathbb{R} \uplus \{\ell\}$, i.e., we suppose $\ell \notin \mathbb{R}$, otherwise the typing rule cannot be applied. Lock $\ell$ is added to $\mathbb{R}$ in rule Acq-C, to ensure that it will be released in the continuation $P$, and in rule New-C, to ensure that a newly created lock is initialised with a release. Correspondingly, rule Rel-C type-checks the release of lock $\ell$ by imposing $\mathbb{R} = \{\ell\}$.

To type-check parallel composition, we use an operation to compose typing environments, written $\Gamma_1 \bullet \Gamma_2$. For this, we set $\emptyset \bullet \Gamma = \Gamma$ and $(\Gamma,\gamma) \bullet \Gamma' = \mathsf{connect}(\gamma;\Gamma \bullet \Gamma')$, where $\mathsf{connect}(\gamma;\{\gamma_1,\ldots,\gamma_k\})$ is undefined as soon as there is $i$ such that $\gamma \cap \gamma_i$ contains at least two distinct elements, and otherwise is defined as

$$\mathsf{connect}(\gamma;\{\gamma_1,\ldots,\gamma_k\}) \quad = \quad \{\gamma_i : \gamma_i\#\gamma\} \uplus \{\gamma \cup \mathsf{flatten}(\{\gamma_i : \gamma_i \cap \gamma \neq \emptyset\})\}.$$

In rule Par-C, we impose that $\mathbb{R}_1$ and $\mathbb{R}_2$ are disjoint: if lock $\ell$ must be released, then this is done either by $P_1$ or by $P_2$. Together with rule Rel-C, this guarantees that any $\ell \in \mathbb{R}$ is released exactly once.

We present some examples to illustrate the type system.

**Example 1.** *Processes $\ell_1.(\overline{\ell_1} \mid \overline{\ell_1})$ and $\ell_1.\ell_2.\overline{\ell_1}$ cannot be typed, because both violate linearity in the usage of locks: the former releases lock $\ell_1$ twice, and the latter does not release $\ell_2$ after acquiring it.*

*Process $P_1 \stackrel{\text{def}}{=} \ell_1.(\overline{\ell_1} \mid \overline{\ell_2})$ acquires lock $\ell_1$, and then releases locks $\ell_1$ and $\ell_2$. Let $\gamma_{12} = \{\ell_1,\ell_2\}$; we can derive $\{\gamma_{12}\};\{\ell_2\} \vdash P_1$: locks $\ell_1$ and $\ell_2$ necessarily belong to the same component when typing $P_1$. Similarly, we have $\{\gamma_{12}\};\{\ell_1\} \vdash P_2$ with $P_2 \stackrel{\text{def}}{=} \ell_2.(\overline{\ell_2} \mid \overline{\ell_1})$. The typing derivations for $P_1$ and $P_2$ cannot be composed, because of the presence of $\gamma_{12}$ in both, so $P_1 \mid P_2$ cannot be typed. This is appropriate, since $P_1 \mid P_2$ presents a typical deadlock situation, where $\ell_1$ is needed to release $\ell_2$ and conversely.*

*On the other hand, process $P_3 \stackrel{\text{def}}{=} \ell_1.(\overline{\ell_1} \mid \overline{\ell_2}) \mid \ell_2.\overline{\ell_2} \mid \ell_1.\overline{\ell_1}$ can be typed: we can derive $\{\gamma_{12}\};\{\ell_2\} \vdash \ell_1.(\overline{\ell_1} \mid \overline{\ell_2})$ and $\{\{\ell_1\},\{\ell_2\}\};\emptyset \vdash \ell_2.\overline{\ell_2} \mid \ell_1.\overline{\ell_1}$, and we can compose these typing derivations, yielding*

$\{\gamma_{12}\};\{\ell_2\} \vdash P_3$. *Crucially, components* $\{\ell_1\}$ *and* $\{\ell_2\}$ *are not merged in the second derivation for the composition to be possible. Using similar ideas, we can define a typable process made of three parallel components* $P_1, P_2, P_3$ *sharing a single lock, say* $\ell$*, as long as each of the* $P_i$ *uses its own locks besides* $\ell$*.*

*We can derive* $\{\gamma_{12}\};\emptyset \vdash P_4$ *with* $P_4 \stackrel{\text{def}}{=} \ell_1.\ell_2.(\overline{\ell_2} \mid \overline{\ell_1})$*. We observe that* $P_4 \mid P_4$ *cannot be typed, although* $P_4 \mid P_4$ *is 'no more deadlocked' than* $P_4$ *alone.*

The typing rules enforce $\mathbb{R} \subseteq \mathrm{dom}(\Gamma)$ when deriving $\Gamma;\mathbb{R} \vdash P$. We say that $\ell$ is *available* in process $P$ if $P$ contains a release of $\ell$ which is not under an acquire on $\ell$ in $P$. Intuitively, when $\Gamma;\mathbb{R} \vdash P$ is derivable, $P$ is a well-typed process in which all lock names in $\mathbb{R}$ are available in $P$. The type system thus guarantees a linearity property on the release of names in $\mathbb{R}$. However, lock names are not *linear names* in the sense of [15], since there can be arbitrarily many acquire operations on a given lock. When all free lock names are available in $P$, i.e. $\Gamma;\mathrm{fln}(P) \vdash P$, we say that $P$ is *complete*.

**Lemma 2.** *The type system enjoys invariance under* $\equiv$ *and subject reduction:* $(i)$ *If* $\Gamma;\mathbb{R} \vdash P$ *and* $P \equiv P'$*, then* $\Gamma;\mathbb{R} \vdash P'$*.* $(ii)$ *If* $\Gamma;\mathbb{R} \vdash P$ *and* $P \to P'$*, then* $\Gamma;\mathbb{R} \vdash P'$ *and* $\mathrm{fln}(P') = \mathrm{fln}(P)$*.*

**Deadlock-Freedom.** Intuitively, a deadlock in CCS$\ell$ arises from an acquire operation that cannot be performed. We say that a *terminated process* is a parallel composition of release operations possibly under some restrictions. A process that contains at least an acquire and cannot reduce is a *stuck process*. So in particular $\ell.\overline{\ell}$ is stuck; the context may provide a release of $\ell$, triggering the acquire on $\ell$. On the other hand, if $P$ is a stuck process and complete, then $P$ is *deadlocked*: intuitively, the context cannot interact with $P$ in order to trigger an acquire operation of $P$. Process $P_{\mathsf{dl}}$ from Section 1 is an example of a deadlock. We show that a complete process can only reduce to a terminated process, avoiding deadlocks.

The proof of deadlock-freedom for CCS$\ell$ provides the structure of the proofs for deadlock-freedom in $\pi\ell$ and leak-freedom in $\pi\ell$w. It relies on progress: any typable process can reduce to reach a terminated process. We first present some lemmas related to the absence of cyclic structures in CCS$\ell$.

**Lemma 3** (Lock-connected processes)**.** *We say that* $P$ *is* lock-connected *if* $\Gamma;\mathbb{R} \vdash P$ *implies* $\Gamma = \Gamma', \gamma$ *for some* $\Gamma', \gamma$*, with* $\mathrm{fln}(P) \subseteq \gamma$*. In this situation, we also have* $\{\gamma\};\mathbb{R} \vdash P$*. If* $P$ *and* $Q$ *are lock-connected and* $\mathrm{fln}(P) \cap \mathrm{fln}(Q)$ *contains at least two distinct names, then* $P \mid Q$ *cannot be typed.*

The property in Lemma 3 does not hold if $P$ and $Q$ are not lock-connected: take for instance $P = Q = \ell_1.\overline{\ell_1} \mid \ell_2.\overline{\ell_2}$, then we can derive $\{\{\ell_1\},\{\ell_2\}\};\emptyset \vdash P \mid Q$. By the typing rule ACQ-C, any process of the form $\ell.P$ is lock-connected. A typical example of a lock-connected process is $\ell_1.(\overline{\ell_1} \mid \overline{\ell_2}) \mid \ell_2.(\overline{\ell_2} \mid \overline{\ell_3})$: here $\gamma = \{\ell_1, \ell_2, \ell_3\}$. Processes similar to this one are used in the following lemma.

**Lemma 4** (No cycle)**.** *We write* $P \stackrel{\ell}{\longleftrightarrow} Q$ *when* $\ell \in \mathrm{fln}(P) \cap \mathrm{fln}(Q)$*. Suppose there are* $k > 1$ *pairwise distinct names* $\ell_1, \ldots, \ell_k$*, and processes* $P_1, \ldots, P_k$ *such that* $\ell_i.P_i \stackrel{\ell_i}{\longleftrightarrow} \ell_{(i+1) \bmod k}.P_{(i+1) \bmod k}$ *for* $1 \leq i \leq k$*. Then* $P_1 \mid \ldots \mid P_k$ *is not typable.*

We use notation $\prod_i P_i$ for the parallel composition of processes $P_i$.

**Lemma 5** (Progress)**.** *If* $\Gamma;\mathrm{fln}(P) \vdash P$*, then either* $P \to P'$ *for some* $P'$*, or* $P \equiv (\nu\widetilde{\ell})\prod_i \overline{\ell_i}$ *where the* $\ell_i$*s are pairwise distinct.*

*Proof.* Write $P \equiv (\nu\widetilde{\ell})P_0$ with $P_0 = \prod_i \overline{\ell_i} \mid \prod_j \ell_j.P_j$. We let $Q_j$ stand for $\ell_j.P_j$, and suppose that there is at least one $Q_j$. We show that under this hypothesis $P_0$ can reduce.

If $\ell_i = \ell_j$ for some $i, j$, then $P_0$ can reduce. We suppose in the following that this is not the case, and consider one of the $Q_j$s. By typing, there exists a unique occurrence of $\ell_j$ available in $P_0$. By hypothesis, this occurrence is not among the $\overline{\ell_i}$s. Therefore, $\ell_j$ is available in $Q_{j'}$ for some unique $j'$ with $j \neq j'$.

We construct a graph having one vertex for each of the $Q_j$s. We draw an edge between $Q_j$ and $Q_{j'}$ when $\ell_j$ is available in $Q_{j'}$. By the reasoning we just made, each vertex is related to at least one other vertex. So the graph necessarily contains a cycle. We can apply Lemma 4 to derive a contradiction.

We make two remarks about the construction of the graph. First, two $Q_j$s may start with an acquire at the same name. The corresponding vertices will have edges leading to the same $Q_{j'}$, and the construction still works. Second, if there is only one $Q_j$, then the available release of $\ell_j$ can synchronise with $Q_j$. □

By Lemma 5, we have that any typable process is not deadlocked. Thus, by subject reduction, we can prove deadlock-freedom.

**Proposition 6** (Deadlock-freedom)**.** *If* $\Gamma;\mathbb{R} \vdash P$ *and* $P \Rightarrow P'$*, then* $P'$ *is not deadlocked.*

**Remark 7.** *As* CCS$\ell$ *is finite, deadlock-freedom ensures that no acquire operation waits forever in a complete typable process, and every complete process reduces to a terminated process: if* $\Gamma;\mathrm{fln}(P) \vdash P$*, then* $P \Rightarrow (\nu\widetilde{\ell})\prod_i \overline{\ell_i}$ *where the* $\ell_i$*s are pairwise distinct.*

## 2.2 $\pi\ell$: Deadlock-Freedom for Higher-Order Locks

**Syntax and Operational Semantics of $\pi\ell$.** $\pi\ell$ extends CCS$\ell$ with the possibility to store *values*, which can be either booleans or locks, in locks. In this sense, $\pi\ell$ features higher-order locks. Processes in $\pi\ell$ are defined as follows:

$$P \quad ::= \quad \ell(\ell').P \mid \overline{\ell}\langle v\rangle \mid (\nu\ell)P \mid P_1 \mid P_2 \mid \mathbf{0} \mid [v = v']P_1, P_2.$$

$v, v'$ denote *values*, defined by $v ::= \ell \mid \mathtt{b}$, where $\mathtt{b} ::= \mathtt{tt} \mid \mathtt{ff}$ is a boolean value. In addition to $\ell, \ell' \ldots$, we sometimes use also $x, y \ldots$ to range over lock names, to suggest a specific usage, like, e.g. in $\ell(x).P$.

Process $\overline{\ell}\langle\ell'\rangle$ is a *release of* $\ell$, and $\ell(\ell').P$ is an *acquire on* $\ell$; we say in both cases that $\ell$ is the *subject* (or that $\ell$ occurs in subject position) and $\ell'$ is the *object*. Restriction and the acquire prefix act as binders, giving rise to the notion of bound and free names. As in CCS$\ell$, we write $\mathrm{fln}(P)$ for the set of free lock names of $P$. $P\{^{v}/_{\ell}\}$ is the process obtained by replacing every free occurrence of $\ell$ with $v$ in $P$. We say that an occurrence of a process $Q$ in $P$ is *guarded* if the occurrence is under an acquire prefix, otherwise it is said *at top-level* in $P$. Additional operators w.r.t. CCS$\ell$ are the inactive process, $\mathbf{0}$, and value comparison: $[v = v']P_1, P_2$ behaves like $P_1$ if values $v$ and $v'$ are equal, and like $P_2$ otherwise.

Structural congruence in $\pi\ell$ is defined by adding the following axioms to $\equiv$ in CCS$\ell$:

$$P \mid \mathbf{0} \equiv P \qquad (\nu\ell)\mathbf{0} \equiv \mathbf{0} \qquad [v = v]P_1, P_2 \equiv P_1 \qquad [v = v']P_1, P_2 \equiv P_2 \text{ if } v \neq v'$$

The last axiom above cannot be used under an acquire prefix: see Appendix A.3 for the definition of $\equiv$. *Execution contexts*, are defined by $E ::= [\cdot] \mid E \mid P \mid (\nu\ell)E$. The axiom for reduction in $\pi\ell$ is:

$$\overline{\overline{\ell}\langle v\rangle \mid \ell(\ell').P \rightarrow P\{^{v}/_{\ell'}\}}$$

$\Rightarrow$ is the reflexive transitive closure of $\rightarrow$. Labelled transitions, written $P \xrightarrow{\mu} P'$, use actions $\mu$ defined by $\mu ::= \ell(v) \mid \overline{\ell}\langle v\rangle \mid \overline{\ell}(\ell') \mid \tau$, and are standard [26]—we recall the definition in Appendix A.3.

$$\textsc{Acq}\ \frac{\Gamma,(\gamma,\ell,\ell');\mathbb{R},\ell\vdash P}{\{\mathsf{flatten}(\Gamma)\uplus(\gamma,\ell)\};\mathbb{R}\vdash\ell(\ell').P} \qquad \textsc{Rel}\ \frac{}{\Gamma,(\gamma,\ell,v);\{\ell\}\vdash\overline{\ell}\langle v\rangle} \qquad \textsc{New}\ \frac{\Gamma,(\gamma,\ell);\mathbb{R},\ell\vdash P}{\Gamma,\gamma;\mathbb{R}\vdash(\nu\ell)P}$$

$$\textsc{Par}\ \frac{\Gamma_1;\mathbb{R}_1\vdash P_1 \qquad \Gamma_2;\mathbb{R}_2\vdash P_2}{\Gamma_1\bullet\Gamma_2;\mathbb{R}_1\uplus\mathbb{R}_2\vdash P_1\mid P_2} \qquad \textsc{Mat}\ \frac{\Gamma;\mathbb{R}\vdash P_1 \qquad \Gamma;\mathbb{R}\vdash P_2}{\Gamma;\mathbb{R}\vdash[v=v']P_1,P_2}$$

Figure 1: Typing rules for $\pi\ell$

**The type system.** We enforce a sorting discipline for names [17], given by $V ::= \mathsf{bool} \mid L$ and $\Sigma(L) = V$: values, that are stored in locks, are either booleans or locks. We consider that all processes we write obey this discipline, which is left implicit. This means for instance that when writing $\overline{\ell}\langle v\rangle$, $\ell$ and $v$ have appropriate sorts; and similarly for $\ell(\ell').P$. In $[v=v']P_1,P_2$, we only compare values with the same sort.

The typing judgement is written $\Gamma;\mathbb{R}\vdash P$, where $\Gamma$ and $\mathbb{R}$ are defined like for CCS$\ell$. We adopt the convention that if $v$ is a boolean value, then $\gamma,v$ is just $\gamma$, and similarly, $\gamma,\ell$ is just $\gamma$ if the sort of $\ell$ is $\mathsf{bool}$. The operation $\Gamma_1\bullet\Gamma_2$ is the same as for CCS$\ell$.

The typing rules for $\pi\ell$ are presented in Figure 1. Again, in rules ACQ and NEW, writing $\mathbb{R},\ell$ imposes $\ell\notin\mathbb{R}$, otherwise the rule cannot be applied. Similarly, the notation $\gamma,\ell,\ell'$ is only defined when $\gamma\#\{\ell,\ell'\}$ and $\ell\neq\ell'$. Rule REL describes the release of a lock containing either a lock or a boolean value: in the latter case, using the convention above, the conclusion of the rule is $\{\{\ell\}\};\{\ell\}\vdash\overline{\ell}\langle\mathsf{b}\rangle$. In rules ACQ and REL, the subject and the object of the operation should belong to the same component. In CCS$\ell$, only prefixing yields such a constraint.

In rule MAT, we do not impose $\{v,v'\}\in\mathrm{dom}(\Gamma)$. A typical example of a process that uses name comparison is $[\ell_1=\ell_2]\overline{\ell}\langle\mathsf{tt}\rangle,\overline{\ell}\langle\mathsf{ff}\rangle$: in this process, $\ell_1$ and $\ell_2$ intuitively represent no threat of a deadlock.

Before presenting the properties of the type system, we make some comments on the discipline it imposes on $\pi$-calculus names when they are used as locks.

**Remark 8** (An acquired lock cannot be stored)**.** *In $\pi\ell$, the obligation to release a lock cannot be transmitted. Accordingly, $\ell'\notin\mathbb{R}=\{\ell\}$ in rule* REL*, and a process like $\ell(\ell').\overline{\ell_1}\langle\ell\rangle$ cannot be typed. We return to this point after Proposition 11.*

**Remark 9** (Typability of higher-order locks)**.** *Locks are a particular kind of names of the asynchronous $\pi$-calculus (A$\pi$). Acquiring a lock that has been stored in another lock boils down to performing a communication in A$\pi$. We discuss how such communications can occur between typed processes.*

*In rule* REL*, $\ell$ and $\ell'$ must belong to the same component of $\Gamma$. So intuitively, if a process contains $\overline{\ell}\langle\ell'\rangle$, this release is the only place where these locks are used 'together'. A reduction involving a well-typed process containing this release therefore looks like*

$$(\overline{\ell}\langle\ell'\rangle\mid P)\ \mid\ (\ell(x).Q\mid Q')\quad\to\quad P\mid Q\{^{\ell'}/_x\}\mid Q'.$$

*Parentheses are used to suggest an interaction between two processes; $\overline{\ell}\langle\ell'\rangle\mid P$ performs the release, and $\ell(x).Q\mid Q'$ performs the acquire. Process P, which intuitively is the continuation of the release, may use locks $\ell$ and $\ell'$, but not together, and similarly for $Q'$. For instance we may have $P=P_\ell\mid P_{\ell'}$, where $\ell'$ does not occur in $P_\ell$, and vice-versa for $P_{\ell'}$. Note also that $\ell'$ is necessarily fresh for $\ell(x).Q'$: otherwise,*

*typability of $\ell(x).Q'$ would impose $\ell$ and $\ell'$ to be in the same component, which would forbid the parallel composition with $\overline{\ell}\langle\ell'\rangle$.*

*Depending on how $P, Q$ and $Q'$ are written, we can envisage several patterns of usages of locks $\ell$ and $\ell'$. A first example is ownership transfer (or delegation): $\ell' \notin \mathrm{fln}(P)$, that is, $P$ renounces usage of $\ell'$. $\ell'$ can be used in $Q$. Note that typing actually also allows $\ell' \in \mathrm{fln}(Q')$, i.e., the recipient already knows $\ell'$.*

*A second possibility could be that $\ell$ is used linearly, in the sense that there is exactly one acquire on $\ell$. In this case, we necessarily have $\ell \notin \mathrm{fln}(P) \cup \mathrm{fln}(Q')$—note that a release of $\ell$ is available in $Q$, by typing. Linearity of $\ell$ means here that exactly one interaction takes place at $\ell$. After that interaction, the release on $\ell$ contained in $Q$ is* inert*, in the sense that no acquire can synchronise with it. We believe that this form of linearity can be used to encode binary session types in an extended version of $\pi\ell$, including variants and polyadicity, along the lines of [14, 3, 4].*

The type system for $\pi\ell$ satisfies the same properties as in CCS$\ell$ (Lemma 2): invariance under structural congruence, merging components and subject reduction. We also have progress and deadlock-freedom:

**Lemma 10** (Progress)**.** *Suppose $\Gamma; \mathrm{fln}(P) \vdash P$, and $P$ is not structurally equivalent to $\mathbf{0}$. Then*

- *either there exists $P'$ such that $P \to P'$,*
- *or $P \equiv (\nu\widetilde{\ell})(\Pi_i \overline{\ell_i} v_i)$ where the $\ell_i$s are pairwise distinct.*

Like in CCS$\ell$, a deadlocked process in $\pi\ell$ is defined as a complete process that is stuck.

**Proposition 11** (Deadlock-freedom)**.** *If $\Gamma; \mathbb{R} \vdash P$ and $P \Rightarrow P'$, then $P'$ is not deadlocked.*

The proof of deadlock-freedom is basically the same as for CCS$\ell$. The reason for that is that although the object part of releases plays a role in the typing rules, it is not relevant to establish progress (Lemma 10). This is the case because in $\pi\ell$, it is not possible to store an acquired lock in another lock (Remark 8).

It seems difficult to extend the type system in order to allow processes that transmit the release obligation on a lock. This would make it possible to type-check, e.g., process $\ell(\ell').\overline{\ell_1}\langle\ell\rangle$, that does not release lock $\ell$ but instead stores it in $\ell_1$. Symmetrically, a process accessing $\ell$ at $\ell_1$ would be in charge of releasing both $\ell_1$ and $\ell$. In such a framework, a process like $(\nu\ell_1)(\overline{\ell_1}\langle\ell\rangle \mid \ell(x).\overline{\ell}\langle x\rangle)$ would be deadlocked, because the inert release $\overline{\ell_1}\langle\ell\rangle$ contains the release obligation on $\ell_1$. The type system in Section 3 makes it possible to transmit the obligation to perform a release (and similarly for a wait).

**Remark 12.** *Similarly to Remark 7, we have that $\Gamma; \mathrm{fln}(P) \vdash P$ implies $P \Rightarrow (\nu\widetilde{\ell})(\Pi_i \overline{\ell_i} v_i)$ where the $\ell_i$s are pairwise distinct. As a consequence, the following holds: if $\Gamma; \mathrm{fln}(P) \vdash P$, then for any $\ell \in \mathrm{fln}(P)$, $P \Rightarrow \xrightarrow{\mu}$, where $\mu$ is a release of $\ell$. This statement would be better suited if infinite computations were possible in $\pi\ell$. We leave the investigation of such an extension of $\pi\ell$ for future work.*

## 2.3 Behavioural Equivalence in $\pi\ell$

We introduce typed barbed equivalence ($\simeq$) and typed bisimilarity ($\approx$) for $\pi\ell$. We show that $\approx$ is a sound technique to establish $\simeq$, and present several examples of (in)equivalences between $\pi\ell$ processes.

### 2.3.1 Barbed Equivalence and Labelled Semantics for $\pi\ell$

A typed relation in $\pi\ell$ is a set of quadruples of the form $(\Gamma, \mathbb{R}, P, Q)$ such that $\Gamma; \mathbb{R} \vdash P$ and $\Gamma; \mathbb{R} \vdash Q$. When a typed relation $\mathscr{R}$ contains $(\Gamma, \mathbb{R}, P, Q)$, we write $\Gamma; \mathbb{R} \vdash P \mathscr{R} Q$. We say that a typed relation $\mathscr{R}$ is symmetric if $\Gamma; \mathbb{R} \vdash P \mathscr{R} Q$ implies $\Gamma; \mathbb{R} \vdash Q \mathscr{R} P$.

Deadlock-freedom has two consequences regarding the definition of barbed equivalence in $\pi\ell$, noted $\simeq$. First, only complete processes should be observed, because intuitively a computation in $\pi\ell$ should not be blocked by an acquire operation that cannot be executed.

Second, Proposition 11 entails that all weak barbs in the sense of A$\pi$ can always be observed in $\pi\ell$. In A$\pi$, a weak barb at $n$ corresponds to the possibility to reduce to a process in which an output at channel $n$ occurs at top-level. We need a stronger notion of barb, otherwise $\simeq$ would be trivial. That behavioural equivalence in $\pi\ell$ is not trivial is shown for instance by the presence of non-determinism. Consider indeed process $P_c \stackrel{\text{def}}{=} (\nu\ell)\big(\ell(x).(\overline{c}\langle x\rangle \mid \overline{\ell}\langle x\rangle) \mid \ell(y).\overline{\ell}\langle \mathsf{ff}\rangle \mid \overline{\ell}\langle \mathsf{tt}\rangle\big)$. Then $P_c \Rightarrow \overline{c}\langle \mathsf{tt}\rangle$ and $P_c \Rightarrow \overline{c}\langle \mathsf{ff}\rangle$ (up to the cancellation of an inert process of the form $(\nu\ell)\overline{\ell}\langle \mathsf{b}\rangle$). We therefore include the object part of releases in barbs. We write $P \downarrow_{\overline{\ell}\langle\ell'\rangle}$ if $P \xrightarrow{\overline{\ell}\langle\ell'\rangle}$, and $P \downarrow_{\overline{\ell}(\nu)}$ if $P \xrightarrow{\overline{\ell}(\ell')}$. We use $\eta$ to range over barbs, writing $P \downarrow_\eta$; the weak version of the predicate, defined as $\Rightarrow\downarrow_\eta$, is written $P \Downarrow_\eta$.

**Definition 13** (Barbed equivalence in $\pi\ell$, $\simeq$)**.** *A symmetric typed relation $\mathscr{R}$ is a* typed barbed bisimulation *if $\Gamma;\mathbb{R} \vdash P\mathscr{R}Q$ implies the three following properties:*

1. *whenever $P,Q$ are complete and $P \to P'$, there is $Q'$ s.t. $Q \Rightarrow Q'$ and $\Gamma;\mathbb{R} \vdash P'\mathscr{R}Q'$;*
2. *for any $\eta$, if $P,Q$ are complete and $P \downarrow_\eta$ then $Q \Downarrow_\eta$;*
3. *for any $E,\Gamma',\mathbb{R}'$ s.t. $\Gamma';\mathbb{R}' \vdash E[P]$ and $\Gamma';\mathbb{R}' \vdash E[Q]$, and $E[P],E[Q]$ are complete, we have $\Gamma';\mathbb{R}' \vdash E[P]\mathscr{R}E[Q]$.*

Typed barbed equivalence*, written $\simeq$, is the greatest typed barbed bisimulation.*

**Lemma 14** (Observing only booleans)**.** *We use $o,o',\ldots$ for lock names that are used to store boolean values. We define $\simeq_o$ as the equivalence defined as in Definition 13, but restricting the second clause to barbs of the form $\downarrow_{\overline{o}\langle \mathsf{b}\rangle}$ and $\Downarrow_{\overline{o}\langle \mathsf{b}\rangle}$. Relation $\simeq_o$ coincides with $\simeq$.*

To define typed bisimilarity, we introduce *type-allowed transitions*. The terminology means that we select among the untyped transitions those that are fireable given the constraints imposed by types.

**Definition 15** (Type-allowed transitions)**.** *When $\Gamma;\mathbb{R} \vdash P$, we write $[\Gamma;\mathbb{R};P] \xrightarrow{\mu} [\Gamma';\mathbb{R}';P']$ if $P \xrightarrow{\mu} P'$ and one of the following holds:*

1. *$\mu = \tau$, in which case $\mathbb{R}' = \mathbb{R}$ and $\Gamma' = \Gamma$;*
2. *$\mu = \overline{\ell}\langle v\rangle$, in which case $(\gamma,\ell,v) \in \Gamma$ for some $\gamma$, and $\mathbb{R}',\ell = \mathbb{R}$, $\Gamma' = \Gamma$;*
3. *$\mu = \overline{\ell}(\ell')$, in which case $\Gamma = \Gamma_0,(\gamma,\ell)$ for some $\Gamma_0,\gamma$, $\Gamma' = \Gamma_0,(\gamma,\ell,\ell')$, and we have $\mathbb{R}',\ell = \mathbb{R},\ell'$;*
4. *$\mu = \ell(v)$, in which case there are $\Gamma_0,\mathbb{R}_0$ s.t. $\Gamma_0;\mathbb{R}_0 \vdash P \mid \overline{\ell}\langle v\rangle$, and $\Gamma' = \Gamma_0,\mathbb{R}' = \mathbb{R}_0$.*

In item 3, $\ell$ is removed from the $\mathbb{R}$ component, and $\ell'$ is added: it is $P'$'s duty to perform the release of $\ell'$, the obligation is not transmitted. An acquire transition involving a higher-order lock merges two distinct components in the typing environment: if $[\Gamma_0,(\gamma,\ell),(\gamma',\ell');\mathbb{R};P] \xrightarrow{\ell(\ell')} [\Gamma';\mathbb{R}';P']$ (item 4 above), then $\Gamma' = \Gamma_0,(\gamma \uplus \gamma' \uplus \{\ell,\ell'\})$ and $\mathbb{R}' = \mathbb{R},\ell$ (and in particular $\ell \notin \mathbb{R}$).

**Lemma 16** (Subject Reduction for type-allowed transitions)**.** *If $[\Gamma;\mathbb{R};P] \xrightarrow{\mu} [\Gamma';\mathbb{R}';P']$, then $\Gamma';\mathbb{R}' \vdash P'$.*

**Definition 17** (Typed bisimilarity, $\approx$)**.** *A typed relation $\mathscr{R}$ is a typed bisimulation if $\Gamma;\mathbb{R} \vdash P\mathscr{R}Q$ implies that whenever $[\Gamma;\mathbb{R};P] \xrightarrow{\mu} [\Gamma';\mathbb{R}';P']$, we have*

1. *either $Q \stackrel{\hat{\mu}}{\Longrightarrow} Q'$ and $\Gamma';\mathbb{R}' \vdash P'\mathscr{R}Q'$ for some $Q'$*

*2. or $\mu$ is an acquire $\ell(v)$, $Q \mid \overline{\ell}\langle v\rangle \Rightarrow Q'$ and $\Gamma';\mathbb{R}' \vdash P' \mathscr{R} Q'$ for some $Q'$,*

*and symmetrically for the type-allowed transitions of $Q$.*

Typed bisimilarity*, written $\approx$, is the largest typed bisimulation.*

We write $\Gamma;\mathscr{R} \vdash P \approx Q$ when $(\Gamma;\mathbb{R},P,Q) \in \approx$. If $\Gamma;\mathbb{R} \vdash P \approx Q$ does not hold, we write $\Gamma;\mathbb{R} \vdash P \not\approx Q$, and similarly for $\Gamma;\mathbb{R} \vdash P \not\simeq Q$.

Proposition 18 below states that relation $\approx$ provides a sound proof technique for $\simeq$. The main property to establish this result is that $\approx$ is preserved by parallel composition: $\Gamma_0;\mathbb{R}_0 \vdash P \approx Q$ implies that for all $T$, whenever $\Gamma;\mathbb{R} \vdash P \mid T$ and $\Gamma;\mathbb{R} \vdash Q \mid T$, we have $\Gamma;\mathbb{R} \vdash P \mid T \approx Q \mid T$.

**Proposition 18** (Soundness)**.** *For any $\Gamma,\mathbb{R},P,Q$, if $\Gamma;\mathbb{R} \vdash P \approx Q$, then $\Gamma;\mathbb{R} \vdash P \simeq Q$.*

The main advantage in using $\approx$ to establish equivalences for $\simeq$ is that we can reason directly on processes, even if they are not complete.

### 2.3.2 Examples of Behavioural Equivalence in $\pi\ell$

**Example 19.** *We discuss some equivalences for $\simeq$.*

*The equivalence $\{\{\ell\}\};\emptyset \vdash \ell(x).\overline{\ell}\langle x\rangle \simeq \mathbf{0}$, which is typical of A$\pi$, holds in $\pi\ell$. This follows directly from the definition of typed bisimilarity, and soundness (Proposition 18).*

*We now let $P \stackrel{\text{def}}{=} \ell(x).(\overline{\ell_0}\langle \mathsf{tt}\rangle \mid \overline{\ell}\langle x\rangle)$ and $Q \stackrel{\text{def}}{=} \overline{\ell_0}\langle \mathsf{tt}\rangle$, and consider whether we can detect the presence of a 'forwarder' at $\ell$ when its behaviour is interleaved with another process. $P$ and $Q$ have different barbs—they are obviously not complete. It turns out that $\{\{\ell,\ell_0\}\};\{\ell_0\} \vdash \ell(x).(\overline{\ell_0}\langle \mathsf{tt}\rangle \mid \overline{\ell}\langle x\rangle) \not\simeq \overline{\ell_0}\langle \mathsf{tt}\rangle$. Indeed, let us consider the context*

$$E \stackrel{\text{def}}{=} [\cdot] \mid \ell_0(y).w(\_).(\overline{w}\langle \mathsf{tt}\rangle \mid \overline{\ell_0}\langle y\rangle) \mid w'(\_).(\overline{w'}\langle \mathsf{tt}\rangle \mid \overline{\ell}\langle v\rangle) \mid \overline{w}\langle \mathsf{ff}\rangle \mid \overline{w'}\langle \mathsf{ff}\rangle,$$

*where $w,w'$ are fresh names and $v$ is a value of the appropriate sort. We have $E[Q] \Rightarrow Q'$ with $Q' \not\downarrow_{\overline{w}\langle \mathsf{ff}\rangle}$ and $Q' \downarrow_{\overline{w'}\langle \mathsf{ff}\rangle}$. On the other hand, for any $P'$ s.t. $E[P] \Rightarrow P'$, if $P' \not\downarrow_{\overline{w}\langle \mathsf{ff}\rangle}$, then $P' \not\downarrow_{\overline{w'}\langle \mathsf{ff}\rangle}$.*

*Contexts like $E$ above make it possible to detect when the process in the hole has some interaction (here, with locks $\ell$ and $\ell_0$).*

*Using similar ideas, we can prove that*

$$\Gamma;\mathbb{R} \vdash \ell_1(x).\ell_2(y).P \not\simeq \ell_2(y).\ell_1(x).P \qquad \text{for appropriate } \Gamma \text{ and } \mathbb{R}.$$

*Indeed, let us define $E_w \stackrel{\text{def}}{=} \overline{w}\langle \mathsf{ff}\rangle \mid w(\_).([\cdot] \mid \overline{w}\langle \mathsf{tt}\rangle)$, where $\_$ stands for an arbitrary lock name, that is not used. We can use the context $[\cdot] \mid E_{w_2}[\overline{\ell_2}\langle v_2\rangle] \mid \overline{\ell_1}\langle v_1\rangle \mid \ell_1(z).E_{w_1}[\overline{\ell_1}\langle z\rangle]$, for fresh names $w_1,w_2$ and appropriate values $v_1,v_2$, to detect the order in which acquires on $\ell_1$ and $\ell_2$ are made.*

In the next two examples, we show equivalences that hold because we work in a typed setting.

**Example 20.** *Suppose $\Gamma;\mathbb{R},\ell \vdash \ell(x).P \mid \ell'(y).(\overline{\ell}\langle v\rangle \mid Q)$. Then we have*

$$\Gamma;\mathbb{R},\ell \vdash \ell(x).P \mid \ell'(y).(\overline{\ell}\langle v\rangle \mid Q) \approx \ell'(y).(\overline{\ell}\langle v\rangle \mid Q \mid \ell(x).P),$$

*because intuitively the acquire on $\ell$ cannot be triggered by the context, due to the presence of a release at $\ell$ in the process. (We remark in passing that $\Gamma;\mathbb{R},\ell \vdash \ell(x).P \mid \ell'(y).(\overline{\ell}\langle v\rangle \mid Q)$ iff $\Gamma;\mathbb{R},\ell \vdash \ell'(y).(\overline{\ell}\langle v\rangle \mid Q \mid \ell(x).P)$, and in this case $\Gamma$ contains a component of the form $(\gamma,\ell,\ell',v)$.)*

*This law can be generalised as follows. We say that $\ell$ is* available *in a context $C$ if the hole does not occur in $C$ neither under a binder for $\ell$, nor under an acquire on $\ell$. So for instance $\ell$ is not available in $(\nu\ell)[\cdot]$, in $\ell_0(\ell).[\cdot]$ or in $\ell(x).[\cdot]$, and $\ell$ is available in $\ell(x).\overline{\ell}\langle x\rangle \mid \overline{\ell}\langle v\rangle \mid [\cdot]$. If $\ell$ is available in $C$, then*

$$\Gamma;\mathbb{R} \vdash \ell(x).P \mid C[\overline{\ell}\langle v\rangle] \approx C[\overline{\ell}\langle v\rangle \mid \ell(x).P] \qquad \text{for appropriate } \Gamma \text{ and } \mathbb{R}.$$

**Example 21.** *Consider the following processes:*

$$\begin{array}{rcl} P_1 & = & (\nu\ell_1)\big(\ell_1.\ell_2.(\overline{\ell_1} \mid \overline{\ell_2}) \mid \ell(x).\ell_1.x.(\overline{\ell_1} \mid \overline{x} \mid \overline{\ell}\langle x\rangle) \mid \overline{\ell_1} \mid \overline{\ell_2}\big) \\ P_2 & = & (\nu\ell_1)\big(\ell_1.\ell_2.(\overline{\ell_1} \mid \overline{\ell_2}) \mid \ell(x).x.\ell_1.(\overline{\ell_1} \mid \overline{x} \mid \overline{\ell}\langle x\rangle) \mid \overline{\ell_1} \mid \overline{\ell_2}\big) \end{array}$$

*Here we use a CCS-like syntax, to ease readability. This notation means that acquire operations are used as forwarders, i.e., the first component of $P_1$ and $P_2$ should be read as $\ell_1(y_1).\ell_2(y_2).(\overline{\ell_1}\langle y_1\rangle \mid \overline{\ell_2}\langle y_2\rangle)$. Moreover, the two releases available at top-level are $\overline{\ell_1}\langle \mathsf{tt}\rangle \mid \overline{\ell_2}\langle \mathsf{tt}\rangle$, and similarly for $\overline{x}\langle \mathsf{tt}\rangle$ (the reasoning also holds if $\ell_1$ and $\ell_2$ are higher-order locks).*

*In the pure $\pi$-calculus, $P_1$ and $P_2$ are not equivalent, because $\ell_2$ can instantiate $x$ in the acquire on $\ell$. We can show $\{\{\ell_2,\ell\}\};\{\ell_2\} \vdash P_1 \approx P_2$ in $\pi\ell$, because the transition $\xrightarrow{\ell(\ell_2)}$ is ruled out by the type system.*

# 3 $\pi\ell$w, a Leak-Free Asynchronous $\pi$-Calculus

## 3.1 Adding Lock Deallocation

$\pi\ell$w is obtained from $\pi\ell$ by adding the *wait construct* $\ell((\ell')).P$ to the grammar of $\pi\ell$. As announced in Section 1, the following reduction rule describes how wait interacts with a release:

$$\frac{}{(\nu\ell)(\overline{\ell}\langle v\rangle \mid \ell((\ell')).P) \to P\{v/\ell'\}}\ \ell \notin \mathrm{fln}(P)$$

The wait instruction deallocates the lock. The continuation may use $\ell'$, the final value of the lock. We say that $\ell((\ell'))$ is a *wait on* $\ell$, and $\ell'$ is bound in $\ell((\ell')).P$.

Types in $\pi\ell$w, written $\mathsf{T},\mathsf{T}',\ldots$, are defined by $\mathsf{T} ::= \mathsf{bool} \mid \langle\mathsf{T}\rangle_{rw}$, and *typing hypotheses* are written $\ell:\mathsf{T}$. In $\ell:\langle\mathsf{T}\rangle_{rw}$, $rw$ is called the *usage* of $\ell$, and $r,w \in \{0,1\}$ are the release and wait *obligations*, respectively, on lock $\ell$. So for instance a typing hypothesis of the form $\ell:\langle\mathsf{T}\rangle_{10}$ means that $\ell$ must be used to perform a release and cannot be used to perform a wait. An hypothesis $\ell:\langle\mathsf{T}\rangle_{00}$ means that $\ell$ can only be used to perform acquire operations. This structure for types makes it possible to transmit the wait and release obligations on a given lock name via higher-order locks.

Our type system ensures that locks are properly deallocated. In contrast to $\pi\ell$, this allows acquired locks to be stored without creating deadlocks. For example, a process like $(\nu\ell_1)(\overline{\ell_1}\langle\ell\rangle \mid \ell(x).\overline{\ell}\langle x\rangle)$ is deadlocked if $\ell_1$ stores the release obligation of $\ell$; however, it cannot be typed as it lacks the wait on $\ell_1$. Adding a wait, e.g. $\ell_1((\ell)).\overline{\ell}\langle v\rangle$ removes the deadlock.

Typing environments have the same structure as in Section 2, except that components $\gamma$ are sets of typing hypotheses instead of simply sets of lock names. $\mathrm{dom}(\Gamma)$ is defined as the set of lock names for which $\Gamma$ contains a typing hypothesis. We write $\Gamma(\ell)=\mathsf{T}$ if the typing hypothesis $\ell:\mathsf{T}$ occurs in $\Gamma$.

We reuse the notation for composition of typing environments. $\Gamma_1 \bullet \Gamma_2$ is defined like in Section 2.1, using the connect operator, to avoid cyclic structures in the sharing of lock names. Additionally, when merging components, we compose typing hypotheses. For any $\ell$, if $\ell:\langle\mathsf{T}_1\rangle_{r_1w_1} \in \mathrm{dom}(\Gamma_1)$ and $\ell:\langle\mathsf{T}_2\rangle_{r_2w_2} \in \mathrm{dom}(\Gamma_2)$, the typing hypothesis for $\ell$ in $\Gamma_1 \bullet \Gamma_2$ is $\ell:\langle\mathsf{T}\rangle_{(r_1+r_2)(w_1+w_2)}$, and is defined only if $\mathsf{T}=\mathsf{T}_1=\mathsf{T}_2$, $r_1+r_2\le 1$ and $w_1+w_2\le 1$.

The typing rules are given in Figure 2. The rules build on the rules for $\pi\ell$, and rely on usages to control the release and wait obligations. In particular, the set $\mathbb{R}$ in Figure 1 corresponds to the set of locks whose usage is of the form $1w$ in this system. To type-check an acquire, we can have usage 00, but also 01, as in, e.g., $\ell(\ell').(\overline{\ell}\langle\ell'\rangle \mid \ell((x)).P)$. In rule REL-W, we impose that all typing hypotheses in $\Gamma_{00}$ (resp. $\gamma_{00}$) have the form $\ell:\langle\mathsf{T}\rangle_{00}$.

$$\textsc{Acq-w}\ \frac{\Gamma,(\gamma,\ell:\langle\mathsf{T}\rangle_{1w},\ell':\mathsf{T})\vdash P}{\{\mathsf{flatten}(\Gamma)\uplus(\gamma,\ell:\langle\mathsf{T}\rangle_{0w})\}\vdash\ell(\ell').P}\qquad \textsc{Rel-w}\ \frac{}{\Gamma_{00},(\gamma_{00},\ell:\langle\mathsf{T}\rangle_{10},v:\mathsf{T})\vdash\overline{\ell}\langle v\rangle}$$

$$\textsc{Wait-w}\ \frac{\{\gamma,\ell':\mathsf{T}\}\vdash P}{\{\gamma,\ell:\langle\mathsf{T}\rangle_{01}\}\vdash\ell((\ell')).P}\qquad \textsc{New-w}\ \frac{\Gamma,(\gamma,\ell:\langle\mathsf{T}\rangle_{11})\vdash P}{\Gamma,\gamma\vdash(\nu\ell)P}\qquad \textsc{Par-w}\ \frac{\Gamma_1\vdash P_1\qquad\Gamma_2\vdash P_2}{\Gamma_1\bullet\Gamma_2\vdash P_1\mid P_2}\qquad \textsc{Nil-w}\ \frac{}{\emptyset\vdash\mathbf{0}}$$

$$\textsc{Mat-w}\ \frac{\Gamma\vdash P_1\qquad\Gamma\vdash P_2}{\Gamma\vdash[v=v']P_1,P_2}$$

Figure 2: Typing rules for $\pi\ell$w

Several notions introduced for the type system of Section 2 have to be adapted in the setting of $\pi\ell$w. While in Section 2 we simply say that a lock $\ell$ is available, here we distinguish whether a release of $\ell$ or a wait on $\ell$ is available. If $P$ has a subterm of the form $\ell((x)).Q$ that does not occur under a binder for $\ell$, we say that a wait on $\ell$ is *available* in $P$. If $\overline{\ell}\langle v\rangle$ occurs in some process $P$ and this occurrence is neither under a binder for $\ell$ nor under an acquire on $\ell$, we say that a release of $\ell$ is *available* in $P$. In addition, a release of $\ell$ (resp. wait on $\ell$) is available in $P$ also if $P$ contains a release of the form $\overline{\ell_0}\langle\ell\rangle$, which does not occur under a binder for $\ell$, and if $\ell$'s type is of the form $\langle\mathsf{T}\rangle_{1w}$ (resp. $\langle\mathsf{T}\rangle_{r1}$).

Like in $\pi\ell$, a deadlocked process in $\pi\ell$w is a complete process that is stuck. The notion of complete process has to be adapted in order to take into account the specificities of $\pi\ell$w. First, the process should not be stuck just because a restriction is missing in order to trigger a name deallocation. Second, we must consider the fact that release and wait obligations can be stored in locks in $\pi\ell$w. As a consequence, when defining complete processes in $\pi\ell$w, we impose some constraints on the free lock names of processes.

In $\pi\ell$w, we say that $\Gamma$ *is complete* if for any $\ell\in\mathrm{dom}(\Gamma)$, either $\Gamma(\ell)=\langle\mathsf{bool}\rangle_{10}$ or $\Gamma(\ell)=\langle\langle\mathsf{T}\rangle_{00}\rangle_{10}$ for some $\mathsf{T}$. To understand this definition, suppose $\Gamma\vdash P$ with $\Gamma$ complete. Then we have, for any free lock name $\ell$ of $P$: $(i)$ the release of $\ell$ is available in $P$; $(ii)$ this release does not carry any obligation; $(iii)$ the wait on $\ell$ is *not* available in $P$. The latter constraint means that if a $P$ contains a wait on some lock, then this lock should be restricted.

The notion of leak-freedom we use is inspired from [12]. In our setting, a situation where some lock $\ell$ is released and will never be acquired again can be seen as a form of memory leak. We say that $P$ *leaks* $\ell$ if $P\equiv(\nu\ell)(P'\mid\overline{\ell}\langle v\rangle)$ with $\ell\notin\mathrm{fln}(P')$. $P$ *has a leak* if $P$ leaks $\ell$ for some $\ell$, and is *leak-free* otherwise.

**Lemma 22** (Progress)**.** *If $\Gamma\vdash P$ and $\Gamma$ is complete, then either $P\to P'$ for some $P'$, or $P\equiv(\nu\widetilde{\ell})(\Pi_i\overline{\ell}_i v_i)$ where the $\ell_i$s are pairwise distinct.*

For lack of space, the proof is presented in Appendix B. Again, it follows the lines of the proof of Lemma 5. To construct a graph containing necessarily a cycle, we associate to every acquire of the form $\ell(x).Q$ an available release of $\ell$, which might occur in a release of the form $\overline{\ell'}\langle\ell\rangle$, if $\ell'$ carries the release obligation. Similarly, to every wait $\ell((x)).Q$, we associate an available release, or, if a release $\overline{\ell}\langle v\rangle$ occurs at top-level, an acquire on $\ell$, that necessarily exists otherwise a reduction could be fired. Finally, using a similar reasoning, to every release of $\ell$ at top-level, we associate a wait on $\ell$, or an acquire on $\ell$.

A consequence of Lemma 22 is that $P\Rightarrow\mathbf{0}$ when $\emptyset\vdash P$.

**Proposition 23** (Deadlock- and Leak-freedom)**.** *$\Gamma \vdash P$ and $P \Rightarrow P'$, then $P'$ neither is deadlocked, nor has a leak.*

**Corollary 24.** *Suppose $\Gamma, \gamma, \ell : \langle \mathsf{bool} \rangle_{10} \vdash P$, and suppose that the usage of all names in $S = \mathrm{dom}(\Gamma, \gamma)$ is 11. Then $(\nu S)P \Downarrow_{\overline{\ell}\langle \mathtt{b} \rangle}$ for some $\mathtt{b}$.*

*Proof.* Immediate by Lemma 22 and subject reduction. □

This property is used to define barbed equivalence below. It does not hold for higher-order locks: simply discarding $x$, the lock stored in $\ell$, might break typability if $\ell$ carries an obligation.

## 3.2 Typed Behavioural Equivalence in $\pi\ell$w

### 3.2.1 Barbed Equivalence

In barbed equivalence in $\pi\ell$ (Definition 13), we compare complete $\pi\ell$ processes, intuitively to prevent blocked acquire operations from making certain observations impossible. Similarly, in $\pi\ell$w, we must also make sure that all wait operations in the processes being observed will eventually be fired. For this, we need to make the process complete (in the sense of Lemma 22), and to add restrictions so that wait transitions are fireable.

However, in order to be able to observe some barbs and discriminate processes, we rely on Corollary 24, and allow names to be unrestricted as long as their type is of the form $\langle \mathsf{bool} \rangle_{10}$. This type means that the lock is first order, and that the context has the wait obligation. In such a situation, interactions at $\ell$ will never be blocked, the whole process is deadlock-free, and eventually reduces to a parallel composition of releases typed with $\langle \mathsf{bool} \rangle_{10}$. Accordingly, we say that a $\pi\ell$w process $P$ is *wait-closed* if $\Gamma \vdash P$ and for any $\ell \in \mathrm{dom}(\Gamma)$, $\Gamma(\ell) = \langle \mathsf{bool} \rangle_{10}$.

A typed relation in $\pi\ell$w is a set of triples $(\Gamma, P, Q)$ such that $\Gamma \vdash P$ and $\Gamma \vdash Q$, and we write $\Gamma \vdash P \mathscr{R} Q$ for $(\Gamma, P, Q) \in \mathscr{R}$. Barbed equivalence in $\pi\ell$w is defined like $\simeq$ (Definition 13), restricting observations to wait-closed processes.

**Definition 25** (Barbed equivalence in $\pi\ell$w, $\simeq_w$)**.** *A symmetric typed relation $\mathscr{R}$ is a* typed barbed bisimulation *if $\Gamma \vdash P \mathscr{R} Q$ implies the three following properties:*

1. *whenever $P, Q$ are wait-closed and $P \to P'$, there is $Q'$ s.t. $Q \Rightarrow Q'$ and $\Gamma \vdash P' \mathscr{R} Q'$;*
2. *if $P, Q$ are wait-closed and $P \downarrow_\eta$ then $Q \Downarrow_\eta$;*
3. *for any $E, \Gamma'$ s.t. $\Gamma' \vdash E[P]$ and $\Gamma' \vdash E[Q]$, and $E[P], E[Q]$ are wait-closed, we have $\Gamma' \vdash E[P] \mathscr{R} E[Q]$.*

Typed barbed equivalence *in $\pi\ell$w, written $\simeq_w$, is the greatest typed barbed bisimulation.*

In the second clause above, $\eta$ can only be of the form $\overline{\ell}\langle \mathtt{b} \rangle$, for some boolean value $\mathtt{b}$. Lemma 14 tells us that we could proceed in the same way when defining $\simeq$.

### 3.2.2 Typed Transitions for $\pi\ell$w, and Bisimilarity

We now define a LTS for $\pi\ell$w. Transitions for name deallocation are not standard in the $\pi$-calculus. To understand how we deal with these, consider $\ell((\ell')).P \mid Q$: this process can do $\xrightarrow{\ell((v))}$ only if $Q$ does not use $\ell$. Similarly, in $\ell((\ell')).P \mid \ell(x).Q \mid \overline{\ell}\langle v \rangle$, the acquire can be fired, and the wait cannot.

Instead of selecting type-allowed transitions among the untyped transitions like in Section 2.3, we give an inductive definition of typed transitions, written $[\Gamma; P] \xrightarrow{\mu} [\Gamma'; P']$. This allows us to use the rules

$$\text{TR}\quad \frac{}{[\{\ell : \langle \mathsf{T}\rangle_{01}, v : \mathsf{T}\}; \overline{\ell}\langle v\rangle] \xrightarrow{\overline{\ell}\langle v\rangle} [\emptyset; \mathbf{0}]}$$

$$\text{TA}\quad \frac{}{[\{\gamma, \ell : \langle \mathsf{T}\rangle_{0w}\}; \ell(\ell').P] \xrightarrow{\ell(v)} [\{\gamma\{^v/_{\ell'}\}, \ell : \langle \mathsf{T}\rangle_{1w}\}; P\{^v/_{\ell'}\}]}$$

$$\text{TW}\quad \frac{}{[\{\gamma, \ell : \langle \mathsf{T}\rangle_{10}\}; \ell((\ell')).P] \xrightarrow{\ell((v))} [\{\gamma\{^v/_{\ell'}\}\}; P\{^v/_{\ell'}\}]}$$

$$\text{TN}\quad \frac{[\Gamma, \gamma, \ell : \langle \mathsf{T}\rangle_{11}; P] \xrightarrow{\mu} [P'; \Gamma', \gamma', \ell : \langle \mathsf{T}\rangle_{11}]}{[\Gamma, \gamma; (\nu\ell)P] \xrightarrow{\mu} [\Gamma', \gamma'; (\nu\ell)P']}\ \ell \notin \mathrm{fln}(\mu)$$

$$\text{TO}\quad \frac{[\Gamma, \gamma, \ell' : \mathsf{T}; P] \xrightarrow{\overline{\ell}\langle \ell'\rangle} [P'; \Gamma', \gamma, \ell' : \mathsf{T}']}{[\Gamma, \gamma; (\nu\ell')P] \xrightarrow{\overline{\ell}(\ell')} [P'; \Gamma', \gamma, \ell' : \mathsf{T}']}$$

$$\text{TT}\quad \frac{[\Gamma, \gamma, \ell : \langle \mathsf{T}\rangle_{11}; P] \xrightarrow{\tau/\ell} [\Gamma', \gamma; P']}{[\Gamma, \gamma; (\nu\ell)P] \xrightarrow{\tau} [\Gamma, \gamma; P']}$$

$$\text{TPC}\quad \frac{[\Gamma_1; P] \xrightarrow{\ell(v)} [\Gamma_1'; P'] \qquad [\Gamma_2; Q] \xrightarrow{\overline{\ell}\langle v\rangle} [\Gamma_2'; Q']}{[\Gamma_1 \bullet \Gamma_2; P \mid Q] \xrightarrow{\tau} [\Gamma_1' \bullet \Gamma_2'; P' \mid Q']}$$

$$\text{TPB}\quad \frac{[\Gamma_1; P] \xrightarrow{\ell(\ell')} [\Gamma_1'; P'] \qquad [\Gamma_2; Q] \xrightarrow{\overline{\ell}(\ell')} [\Gamma_2'; Q']}{[\Gamma_1 \bullet \Gamma_2; P \mid Q] \xrightarrow{\tau} [\Gamma_1' \bullet \Gamma_2'; (\nu\ell')(P' \mid Q')]}$$

$$\text{TPP}\quad \frac{[\Gamma_1; P] \xrightarrow{\mu} [\Gamma_1'; P'] \qquad \Gamma_2 \vdash Q}{[\Gamma_1 \bullet \Gamma_2; P \mid Q] \xrightarrow{\mu} [\Gamma_1' \bullet_\mu \Gamma_2; P' \mid Q]}$$

$$\text{TPT}\quad \frac{[\Gamma_1; P] \xrightarrow{\ell((v))} [\Gamma_1'; P'] \qquad [\Gamma_2; Q] \xrightarrow{\overline{\ell}\langle v\rangle} [\Gamma_2'; Q']}{[\Gamma_1 \bullet \Gamma_2; P \mid Q] \xrightarrow{\tau/\ell} [\Gamma_1' \bullet_{\tau/\ell} \Gamma_2'; P' \mid Q']}$$

$$\text{TPTB}\quad \frac{[\Gamma_1; P] \xrightarrow{\ell((v))} [\Gamma_1'; P'] \qquad [\Gamma_2; Q] \xrightarrow{\overline{\ell}(\ell')} [\Gamma_2'; Q']}{[\Gamma_1 \bullet \Gamma_2; P \mid Q] \xrightarrow{\tau/\ell} [\Gamma_1' \bullet_{\tau/\ell} \Gamma_2'; (\nu\ell')(P' \mid Q')]}$$

Figure 3: $\pi\ell$w, Typed LTS. We omit symmetric versions of rules involving parallel compositions

for parallel composition in order to control the absence of a lock, when a lock deallocation is involved. Technically, this is done by refining the definition of the operator to compose typing contexts.

Actions of the LTS are defined as follows: $\mu ::= \ell(v) \mid \overline{\ell}\langle v\rangle \mid \overline{\ell}(\ell') \mid \tau \mid \ell((v)) \mid \tau/\ell$. Name $\ell$ plays a particular role in transitions along wait actions $\ell((v))$ and *wait synchronisations* $\tau/\ell$: since $\ell$ is deallocated, we must make sure that it is not used elsewhere in the process. We define $\Gamma_1 \bullet_\mu \Gamma_2$ as being equal to $\Gamma_1 \bullet \Gamma_2$, with the additional constraint that $\ell \notin \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2)$ when $\mu = \ell((v))$ or $\mu = \tau/\ell$, otherwise $\Gamma_1 \bullet_\mu \Gamma_2$ is not defined. The rules defining the LTS are given on Figure 3. We define $\mathrm{fln}(\overline{\ell}(\ell')) = \mathrm{fln}(\tau/\ell) = \{\ell\}$, and $\mathrm{fln}(\ell(v)) = \mathrm{fln}(\ell((v))) = \mathrm{fln}(\overline{\ell}\langle v\rangle) = \{\ell, v\}$ (with the convention that $\{\ell, v\} = \{\ell\}$ if $v$ is a boolean value).

We comment on the transition rules. Rules TR, TA and TW express the meaning of usages (respectively, 01, $0w$ and 10). In rule TT, $\ell$ is deallocated, and the restriction on $\ell$ is removed. In rules TPT, TPTB we rely on operation $\Gamma_1 \bullet_\mu \Gamma_2$ to make sure that $\ell$ does not appear in both parallel components of the continuation process, and similarly for TPP in the case where $\mu$ involves deallocation of $\ell$.

Typability is preserved by typed transitions: if $\Gamma \vdash P$ and $[\Gamma; P] \xrightarrow{\mu} [\Gamma'; P']$, then $\Gamma' \vdash P'$.

Bisimilarity in $\pi\ell$w takes into account the additional transitions w.r.t. $\pi\ell$, and is sound for $\simeq_w$.

**Definition 26** (Typed Bisimilarity in $\pi\ell$w, $\approx_w$)**.** *A typed relation $\mathscr{R}$ is a typed bisimulation if* $\Gamma \vdash P \mathscr{R} Q$

*implies that whenever* $[\Gamma;P] \xrightarrow{\mu} [\Gamma';P']$*, we have*

1. *either* $Q \stackrel{\hat{\mu}}{\Rightarrow} Q'$ *and* $\Gamma' \vdash P' \mathscr{R} Q'$ *for some* $Q'$
2. *or* $\mu$ *is an acquire* $\ell(v)$*,* $Q \mid \overline{\ell}\langle v\rangle \Rightarrow Q'$ *and* $\Gamma' \vdash P' \mathscr{R} Q'$ *for some* $Q'$*,*
3. *or* $\mu$ *is a wait* $\ell((v))$*,* $(\nu\ell)(Q \mid \overline{\ell}\langle v\rangle) \Rightarrow Q'$ *and* $\Gamma' \vdash P' \mathscr{R} Q'$ *for some* $Q'$*,*
4. *or* $\mu = \tau/\ell$*,* $(\nu\ell)Q \Rightarrow Q'$ *and* $\Gamma' \vdash P' \mathscr{R} Q'$ *for some* $Q'$*,*

*and symmetrically for the typed transitions of* $Q$*.* Typed bisimilarity in $\pi\ell$w*, written* $\approx_w$*, is the largest typed bisimulation.*

**Proposition 27** (Soundness)**.** *For any* $\Gamma, P, Q$*, if* $\Gamma \vdash P \approx_w Q$*, then* $\Gamma \vdash P \simeq_w Q$*.*

**Example 28.** *The law* $\ell(x).\overline{\ell}\langle x\rangle = \mathbf{0}$ *holds in* $\pi\ell$w*, at type* $\ell : \langle \mathsf{T} \rangle_{00}$*, for any* $\mathsf{T}$*.*
*Suppose* $\Gamma \vdash \ell(x).P \mid \ell((y)).Q$*. Then we can prove* $\Gamma \vdash \ell(x).P \mid \ell((y)).Q \approx_w \ell(x).(P \mid \ell((y)).Q)$*.*
*Using this equivalence and the law of asynchrony, we can deduce* $\ell((x)).P \simeq_w \ell(x).(\overline{\ell}\langle x\rangle \mid \ell((x)).P)$*.*

An equivalence between $\pi\ell$ processes is also valid in $\pi\ell$w. To state this property, given $P$ in $\pi\ell$, we introduce $[\![P]\!]_w$, its translation in $\pi\ell$w. The definition of $[\![P]\!]_w$ is simple, as we just need to add wait constructs under restrictions for $[\![P]\!]_w$ to be typable.

**Lemma 29.** *Suppose* $\Gamma;\mathbb{R} \vdash P \approx Q$*. Then* $\Gamma_w \vdash [\![P]\!]_w \approx_w [\![Q]\!]_w$ *for some* $\pi\ell$w *typing environment* $\Gamma_w$*.*

This result shows that the addition of wait does not increase the discriminating power of contexts. We refer to Appendix B for the definition of $[\![P]\!]_w$ and a discussion of the proof of Lemma 29.

## 4 Related and Future Work

The basic type discipline for lock names that imposes a safe usage of locks by always releasing a lock after acquiring it is discussed in [13]. This is specified using *channel usages* (not to be confused with the usages of Section 3.1). Channel usages in [13] are processes in a subset of CCS, and can be defined in sophisticated ways to control the behaviour of $\pi$-calculus processes. The encoding of references in the asynchronous $\pi$-calculus studied in [7] is also close to how locks are used in $\pi\ell$w. A reference is indeed a lock that must be released *immediately* after the acquire. The typed equivalence to reason about reference names in [7] has important differences w.r.t. $\simeq_w$, notably because the deadlock- and leak-freedom properties are not taken into consideration in that work.

The type system for $\pi\ell$w has several ideas in common with [12]. That paper studies $\lambda_{\text{lock}}$, a functional language with higher-order locks and thread spawning. The type system for $\lambda_{\text{lock}}$ guarantees leak- and deadlock-freedom by relying on duality and linearity properties, which entail the absence of cycles. In turn, this approach originates in work on binary session types, and in particular on concurrent versions of the Curry-Howard correspondence [10, 6, 28, 2, 27, 22].

$\pi\ell$w allows a less controlled form of interaction than functional languages or binary sessions. Important differences are: names do not have to be used linearly; there is no explicit notion of thread, neither a fork instruction, in $\pi\ell$w; reduction is not deterministic. The type system for $\pi\ell$w controls parallel composition to rule out cyclic structures among interacting processes.

The simplicity of the typing rules, and of the proofs of deadlock- and leak-freedom, can be leveraged to develop a theory of typed behavioural equivalence for $\pi\ell$ and $\pi\ell$w. Soundness of bisimilarity provides a useful tool to establish equivalence results. Proving completeness is not obvious, intuitively because

the constraints imposed by typing prevent us from adapting standard approaches. The way $\approx_w$ is defined should allow us to combine locks with other programming constructs in order to reason about programs featuring locks and, e.g., functions, continuations, and references. Work in this direction will build on [19, 23, 5, 8, 21].

Our proofs of deadlock- and leak-freedom suggest that there is room for a finer analysis of how lock names are used. It is natural to try and extend our type system in order to accept more processes, while keeping the induced behavioural equivalence tractable. A possibility for this is to add *lock groups* [12], with the aim of reaching an expressiveness comparable to the system in [12]. In a given lock group, locks are ordered, which makes it possible to analyse systems having a cyclic topology.

Relying on orders to program with locks is a natural approach, that has been used to define expressive type systems for lock freedom in the $\pi$-calculus [11, 13, 20]. In these works, some labelling is associated to channels or to actions on channels, and the typing rules guarantee that it is always possible to define an order, yielding lock-freedom. We plan to study how our type system can be extended with lock groups or ideas from type systems based on orders.

Rule (1) from Section 1 explains in a concise way how the wait operation behaves. Part of the difficulty in Section 3 is in defining a labelled semantics that is compatible with the 'magic' of executing a wait on $\ell$ only when the restriction can be put on top of the final release of $\ell$. We plan to provide a more operational description of deallocation, using, e.g., reference counting as in [12]. $\pi\ell$w could then be seen as a language to describe at high-level what happens at a lower level when using and deallocating locks.

**Acknowledgement.** We are grateful to Jules Jacobs for an interesting discussion about this work, and for suggesting the reduction rule (1) from Section 1. We also thank the anonymous referees for their helpful remarks and advices.

## References

[1] Roberto M. Amadio, Ilaria Castellani & Davide Sangiorgi (1998): *On Bisimulations for the Asynchronous pi-Calculus*. *Theor. Comput. Sci.* 195(2), pp. 291–324, doi:10.1016/S0304-3975(97)00223-5.

[2] Luís Caires & Frank Pfenning (2010): *Session Types as Intuitionistic Linear Propositions*. In Paul Gastin & François Laroussinie, editors: *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, *Lecture Notes in Computer Science* 6269, Springer, pp. 222–236, doi:10.1007/978-3-642-15375-4_16.

[3] Ornela Dardha, Elena Giachino & Davide Sangiorgi (2017): *Session types revisited*. *Inf. Comput.* 256, pp. 253–286, doi:10.1016/j.ic.2017.06.002.

[4] Ornela Dardha, Elena Giachino & Davide Sangiorgi (2022): *Session Types Revisited: A Decade Later*. In: *PPDP 2022: 24th International Symposium on Principles and Practice of Declarative Programming, Tbilisi, Georgia, September 20 - 22, 2022*, ACM, pp. 12:1–12:4, doi:10.1145/3551357.3556676.

[5] Adrien Durier, Daniel Hirschkoff & Davide Sangiorgi (2018): *Eager Functions as Processes*. In Anuj Dawar & Erich Grädel, editors: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, ACM, pp. 364–373, doi:10.1145/3209108.3209152.

[6] Simon J. Gay & Vasco Thudichum Vasconcelos (2010): *Linear type theory for asynchronous session types*. *J. Funct. Program.* 20(1), pp. 19–50, doi:10.1017/S0956796809990268.

[7] Daniel Hirschkoff, Enguerrand Prebet & Davide Sangiorgi (2020): *On the Representation of References in the Pi-Calculus*. In Igor Konnov & Laura Kovács, editors: *31st International Conference on Concurrency Theory, CONCUR 2020, LIPIcs* 171, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 34:1–34:20, doi:10.4230/LIPIcs.CONCUR.2020.34.

[8] Daniel Hirschkoff, Enguerrand Prebet & Davide Sangiorgi (2021): *On sequentiality and well-bracketing in the π-calculus*. In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, IEEE, pp. 1–13, doi:10.1109/LICS52264.2021.9470559.

[9] Kohei Honda & Mario Tokoro (1991): *An Object Calculus for Asynchronous Communication*. In Pierre America, editor: *ECOOP'91 European Conference on Object-Oriented Programming, Geneva, Switzerland, July 15-19, 1991, Proceedings*, *Lecture Notes in Computer Science* 512, Springer, pp. 133–147, doi:10.1007/BFb0057019.

[10] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In Chris Hankin, editor: *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, *Lecture Notes in Computer Science* 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.

[11] Atsushi Igarashi & Naoki Kobayashi (2001): *A generic type system for the Pi-calculus*. In Chris Hankin & Dave Schmidt, editors: *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, ACM, pp. 128–141, doi:10.1145/360204.360215.

[12] Jules Jacobs & Stephanie Balzer (2023): *Higher-Order Leak and Deadlock Free Locks*. *Proc. ACM Program. Lang.* 7(POPL), pp. 1027–1057, doi:10.1145/3571229.

[13] Naoki Kobayashi (2002): *Type Systems for Concurrent Programs*. In Bernhard K. Aichernig & T. S. E. Maibaum, editors: *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002, Revised Papers*, *Lecture Notes in Computer Science* 2757, Springer, pp. 439–453, doi:10.1007/978-3-540-40007-3_26.

[14] Naoki Kobayashi (2007): *Type Systems for Concurrent Programs*. Extended version of [13].

[15] Naoki Kobayashi, Benjamin C. Pierce & David N. Turner (1999): *Linearity and the pi-calculus*. *ACM Trans. Program. Lang. Syst.* 21(5), pp. 914–947, doi:10.1145/330249.330251.

[16] Massimo Merro & Davide Sangiorgi (2004): *On asynchrony in name-passing calculi*. *Math. Struct. Comput. Sci.* 14(5), pp. 715–767, doi:10.1017/S0960129504004323.

[17] R. Milner (1991): *The polyadic π-calculus: a tutorial*. Technical Report ECS–LFCS–91–180, LFCS. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.

[18] Robin Milner (1980): *A Calculus of Communicating Systems*. *Lecture Notes in Computer Science* 92, Springer, doi:10.1007/3-540-10235-3.

[19] Robin Milner (1992): *Functions as Processes*. *Math. Struct. Comput. Sci.* 2(2), pp. 119–141, doi:10.1017/S0960129500001407.

[20] Luca Padovani (2014): *Deadlock and lock freedom in the linear π-calculus*. In Thomas A. Henzinger & Dale Miller, editors: *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 '14, Vienna, Austria, July 14 - 18, 2014*, ACM, pp. 72:1–72:10, doi:10.1145/2603088.2603116.

[21] Enguerrand Prebet (2022): *Functions and References in the Pi-Calculus: Full Abstraction and Proof Techniques*. In Mikolaj Bojanczyk, Emanuela Merelli & David P. Woodruff, editors: *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, *LIPIcs* 229, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 130:1–130:19, doi:10.4230/LIPIcs.ICALP.2022.130.

[22] Pedro Rocha & Luís Caires (2023): *Safe Session-Based Concurrency with Shared Linear State*. In Thomas Wies, editor: *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*, *Lecture Notes in Computer Science* 13990, Springer, pp. 421–450, doi:10.1007/978-3-031-30044-8_16.

[23] Davide Sangiorgi (1994): *The Lazy Lambda Calculus in a Concurrency Scenario*. *Inf. Comput.* 111(1), pp. 120–153, doi:10.1006/inco.1994.1042.

[24] Davide Sangiorgi (1996): *pi-Calculus, Internal Mobility, and Agent-Passing Calculi*. *Theor. Comput. Sci.* 167(1&2), pp. 235–274, doi:10.1016/0304-3975(96)00075-8.

[25] Davide Sangiorgi (1997): *The Name Discipline of Uniform Receptiveness (Extended Abstract)*. In Pierpaolo Degano, Roberto Gorrieri & Alberto Marchetti-Spaccamela, editors: *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings*, *Lecture Notes in Computer Science* 1256, Springer, pp. 303–313, doi:10.1007/3-540-63165-8_187.

[26] Davide Sangiorgi & David Walker (2001): *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press.

[27] Bernardo Toninho, Luís Caires & Frank Pfenning (2013): *Higher-Order Processes, Functions, and Sessions: A Monadic Integration*. In Matthias Felleisen & Philippa Gardner, editors: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, *Lecture Notes in Computer Science* 7792, Springer, pp. 350–369, doi:10.1007/978-3-642-37036-6_20.

[28] Philip Wadler (2014): *Propositions as sessions*. *J. Funct. Program.* 24(2-3), pp. 384–418, doi:10.1017/S095679681400001X.

# A Additional Material for Section 2

## A.1 CCS$\ell$, Operational Semantics

Structural congruence is the least congruence satisfying the following axioms:

$$\overline{P \mid Q \equiv Q \mid P} \qquad \overline{P \mid (Q \mid R) \equiv (P \mid Q) \mid R}$$

$$\overline{P \mid (\nu\ell)Q \equiv (\nu\ell)(P \mid Q)} \text{ if } \ell \notin \mathrm{fln}(P) \qquad \overline{(\nu\ell)(\nu\ell')P \equiv (\nu\ell')(\nu\ell)P}$$

To define reduction, we introduce *execution contexts*, $E$, given by $E ::= [\cdot] \mid E \mid P \mid (\nu\ell)E$, where $[\cdot]$ is the hole. $E[P]$ is the process obtained by replacing the hole in $E$ with $P$.

Reduction is defined by the following rules:

$$\overline{\overline{\ell} \mid \ell.P \to P} \qquad \frac{P \to P'}{E[P] \to E[P']} \qquad \frac{Q \equiv P \qquad P \to P' \qquad P' \equiv Q'}{Q \to Q'}$$

## A.2 CCS$\ell$, Properties of the Type System

*of Lemma 4.* We show by induction on $k$ that $\ell_1.P_1 \mid \ldots \mid \ell_{k-1}.P_{k-1}$ is lock-connected: this holds because for every $i$, $\ell_i.P_i$ is lock-connected, and because $\ell_i.P_i \overset{\ell_i}{\longleftrightarrow} \ell_{i+1}.P_{i+1}$ for all $i < k$.

Moreover, we know $\ell_{k-1}.P_{k-1} \overset{\ell_{k-1}}{\longleftrightarrow} \ell_k.P_k$ and $\ell_k.P_k \overset{\ell_k}{\longleftrightarrow} \ell_1.P_1$. So names $\ell_{k-1}$ and $\ell_k$ belong to the free names both of $\ell_1.P_1 \mid \ldots \mid \ell_{k-1}.P_{k-1}$ and of $\ell_k.P_k$. By Lemma 3, this prevents $\ell_1.P_1 \mid \ldots \mid \ell_k.P_k$ from being typable. □

### A.3 $\pi\ell$, Operational Semantics

Structural congruence in $\pi\ell$, written $\equiv$, is standard, except for the treatment of mismatch. Indeed, the corresponding axiom cannot be used under an acquire prefix.

To handle this, we introduce an auxiliary structural congruence relation, written $\equiv_{\mathrm{r}}$. Relation $\equiv$ is the smallest equivalence relation that satisfies the axioms for $\equiv$ in CCS$\ell$, plus the following ones

$$\textsc{PNil}\ \frac{}{P \mid \mathbf{0} \equiv P} \qquad \textsc{RNil}\ \frac{}{(\nu\ell)\mathbf{0} \equiv \mathbf{0}} \qquad \textsc{Mat}\ \frac{}{[v = v]P_1, P_2 \equiv P_1} \qquad \textsc{Mis}\ \frac{}{[v = v']P_1, P_2 \equiv P_2}\ \text{if } v \neq v'$$

and also the contextual axioms

$$\textsc{CPar}\ \frac{P \equiv Q}{P \mid T \equiv Q \mid T} \qquad \textsc{CRes}\ \frac{P \equiv Q}{(\nu\ell)P \equiv (\nu\ell)Q} \qquad \textsc{CAcq}\ \frac{P \equiv_{\mathrm{r}} Q}{\ell(\ell').P \equiv \ell(\ell').Q}$$

The last axiom refers to $\equiv_{\mathrm{r}}$, which is defined like $\equiv$, except that Mis is omitted and CAcq is replaced by

$$\textsc{CAcq}_{\mathrm{r}}\ \frac{P \equiv_{\mathrm{r}} Q}{\ell(\ell').P \equiv_{\mathrm{r}} \ell(\ell').Q}$$

**Labelled Semantics for $\pi\ell$.** Actions of the LTS are defined by $\mu \quad ::= \quad \ell(v) \mid \overline{\ell}\langle v\rangle \mid \overline{\ell}(\ell') \mid \tau$.

The set of free names of $\mu$ is defined by $\mathrm{fln}(\overline{\ell}\langle v\rangle) = \mathrm{fln}(\ell(v)) = \{\ell, v\}$, $\mathrm{fln}(\tau) = \emptyset$ and $\mathrm{fln}(\overline{\ell}(\ell')) = \{\ell\}$.

The set of bound names of $\mu$ is defined by $\mathrm{bln}(\mu) = \emptyset$, except for $\mathrm{bln}(\overline{\ell}(\ell')) = \{\ell'\}$.

The transition rules are the following:

$$\frac{}{\overline{\ell}\langle v\rangle \xrightarrow{\overline{\ell}\langle v\rangle} \mathbf{0}} \qquad \frac{}{\ell(\ell').P \xrightarrow{\ell(v)} P\{v/\ell'\}} \qquad \frac{P \xrightarrow{\overline{\ell}\langle\ell'\rangle} P'}{(\nu\ell')P \xrightarrow{\overline{\ell}(\ell')} P'} \qquad \frac{P \xrightarrow{\mu} P'}{(\nu\ell)P \xrightarrow{\mu} (\nu\ell)P'}\ \ell \notin \mathrm{fln}(\mu)$$

$$\frac{P \xrightarrow{\mu} P'}{P \mid Q \xrightarrow{\mu} P' \mid Q}\ \mathrm{fln}(Q) \cap \mathrm{bln}(\mu) = \emptyset \qquad \frac{P \xrightarrow{\overline{\ell}\langle v\rangle} P' \qquad Q \xrightarrow{\ell(v)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \frac{P \xrightarrow{\overline{\ell}(\ell')} P' \qquad Q \xrightarrow{\ell(\ell')} Q'}{P \mid Q \xrightarrow{\tau} (\nu\ell')(P' \mid Q')}$$

$$\frac{P_1 \xrightarrow{\mu} P_1'}{[v = v]P_1, P_2 \xrightarrow{\mu} P_1'} \qquad \frac{P_2 \xrightarrow{\mu} P_2'}{[v = v']P_1, P_2 \xrightarrow{\mu} P_2'}\ v \neq v'$$

## B Additional Material from Section 3

### B.1 Leak-Freedom in $\pi\ell$w

The proof of Lemma 22 follows the approach of the proof of Lemma 5. An additional difficulty with respect to the latter proof is that release and wait obligations on a given lock need not be explicit in the process, in the sense that they can be stored in another lock.

*Proof.* We first consider the situation where $\Gamma \vdash P_0$, $\Gamma$ is complete, and we can write

$$P_0 \equiv (\nu S)(\prod_i \overline{\ell_i}\langle v_i\rangle \mid \prod_j \ell_j(x_j).P_j \mid \prod_k \ell_k((y_k)).Q_k).$$

We let $P = \prod_i \overline{\ell_i}\langle v_i\rangle \mid \prod_j \ell_j(x_j).P_j \mid \prod_k \ell_k((y_k)).Q_k$.

We introduce some terminology to reason about this decomposition. A *prime process* is a process of the form $\overline{\ell}\langle v\rangle$, $\ell(x).P'$ or $\ell((y)).P'$. Here "prime" refers to the fact that such processes cannot be decomposed modulo $\equiv$. We call *subject* of a prime process the name that occurs in subject position in the topmost prefix of that process: these are the $\ell_i$s, the $\ell_j$s and the $\ell_k$s in the decomposition above.

We make the two following observations. First, for any $\ell \in \mathrm{fln}(P)$, either $\ell \in \mathrm{fln}(P_0)$, or a release of $\ell$ and a wait on $\ell$ must be available, by typing. Second, none of the $\ell_i$ is equal to one of the $\ell_j$, since otherwise $P_0$ could reduce. Moreover, if some $\ell_i$ is equal to one of the $\ell_k$s, then $P$ necessarily contains an acquire on $\ell_i$, since otherwise $P_0$ could reduce by performing a wait transition. In the following, we do not consider the prime processes whose subject is in $\Gamma$. Recall that these processes are outputs $\overline{\ell_i}\langle v_i\rangle$ with $v_i$ being either of type $\mathsf{bool}$ or $\langle \mathsf{T}\rangle_{00}$.

To derive a contradiction, we show that the subject of every prime process occurs free in another prime process having a different subject. We examine the three forms of prime processes.

- Consider first $\ell_j(x_j).P_j$. The available *release of* $\ell_j$ cannot occur at top-level, since otherwise $P$ could reduce. The release cannot be available under an acquire or wait prefix on $\ell_j$, by typing and by definition of being available.

  The release of $\ell_j$ may be available in one of the $P_{j'}$s, or in one of the $P_k$s occurring under a prefix at some lock name different from $\ell_j$. In both cases, $\ell_j$ occurs in another prime process having a different subject.

  If the release on $\ell_j$ is available neither in the $P_j$s nor in the $P_k$s, then there exists another release of the form $\overline{\ell}\langle \ell_j\rangle$ for some $\ell$, that does not occur under an acquire on $\ell_j$. We remark that $\ell$'s usage is of the form $1w$, and that $\ell \neq \ell_j$.

  Thus, the release of $\ell$ necessarily occurs in a prime process whose subject is different from $\ell_j$.

- Consider now $\ell_k((y_k)).P_k$. As above, we reason about the *release of* $\ell_k$. The only difference is that the release of $\ell_k$ may occur at top-level. If this is the case, then there is necessarily an acquire on $\ell_k$, otherwise $P$ could reduce. This acquire cannot occur at top-level, since otherwise $P$ could reduce, by performing a wait transition. Hence, there is a prime process whose subject is different from $\ell_k$ that contains an acquire on $\ell_k$.

- Consider $\overline{\ell_i}\langle v_i\rangle$. We reason about the *wait on* $\ell_i$. If the wait on $\ell_i$ occurs at top-level, then, as above, an acquire on $\ell_i$ must occur in $P$, since otherwise $P_0$ could reduce. That acquire on $\ell_i$ cannot occur at top-level, since otherwise $P$ could reduce. So in this case $\ell_i$ occurs in a prime process whose subject is different from $\ell_i$.

  If the wait on $\ell_i$ does not occur at top-level, then it can occur in a prime process whose subject is different from $\ell_i$: that process cannot start with an acquire on $\ell_i$ since otherwise $P$ could reduce.

  The last possibility is that a subterm of the form $\overline{\ell}\langle \ell_i\rangle$ occurs in some other prime process, and $\ell$ carries the wait obligation. Reasoning as above, the subject of the prime process cannot be $\ell_i$.

We have shown that every prime process in the decomposition above whose subject is $\ell$ can be connected with a different prime process. Like in the proofs of deadlock-freedom, we obtain a cycle, which is impossible by (the counterpart of) Lemma 4.

□

## B.2 Translating a $\pi\ell$ Process in $\pi\ell$w

If $P$ is a $\pi\ell$ process, $[\![P]\!]_w$ is its translation into $\pi\ell$w, defined as follows:

$$[\![(\nu\ell)P]\!]_w = (\nu\ell)([\![P]\!]_w \mid \ell((x)).\mathbf{0}) \qquad [\![\ell(\ell').P]\!]_w = \ell(\ell').[\![P]\!]_w \qquad [\![\overline{\ell}\langle v\rangle]\!]_w = \overline{\ell}\langle v\rangle$$

$$[\![P_1 \mid P_2]\!]_w = [\![P_1]\!]_w \mid [\![P_2]\!]_w \qquad [\![[v = v']P_1, P_2]\!]_w = [v = v'][\![P_1]\!]_w, [\![P_2]\!]_w$$

To prove Lemma 29, we establish a correspondence between typing in $\pi\ell$ and in $\pi\ell$w. If $\Gamma;\mathbb{R} \vdash P$, the typing environment to type $P$ seen as a $\pi\ell$w process is constructed by making sorts explicit, and by assigning usage 10 for name $\ell$ if $\ell \in \mathbb{R}$, and 00 otherwise. Conversely, if $P \in \pi\ell$ can be typed as a $\pi\ell$w process with $\Gamma_w \vdash P$, then we can suppose that $\Gamma_w$ does not contain any usage of the form $r1$. We recover a $\pi\ell$ typing for $P$ by collecting all names having type usage 10 in $\mathbb{R}$, and erasing type information in the components of $\Gamma_w$, yielding $\Gamma$, so that $\Gamma;\mathbb{R} \vdash P$.

This correspondence is extended to a correspondence between transitions, so that a bisimulation relation in $\pi\ell$ is also a bisimulation in $\pi\ell$w, via the aforementioned translation. To prove the latter property, we rely on the equivalence $\{\{v\}\} \vdash (\nu\ell)(\overline{\ell}\langle v\rangle \mid \ell((x)).\mathbf{0}) \approx_w \mathbf{0}$ in $\pi\ell$w.

# Deriving Abstract Interpreters from Skeletal Semantics

Thomas Jensen
INRIA, Rennes
thomas.jensen@inria.fr

Vincent Rébiscoul
Université de Rennes, Rennes
vincent.rebiscoul@inria.fr

Alan Schmitt
INRIA, Rennes
alan.schmitt@inria.fr

This paper describes a methodology for defining an executable abstract interpreter from a formal description of the semantics of a programming language. Our approach is based on Skeletal Semantics and an abstract interpretation of its semantic meta-language. The correctness of the derived abstract interpretation can be established by compositionality provided that correctness properties of the core language-specific constructs are established. We illustrate the genericness of our method by defining a Value Analysis for a small imperative language based on its skeletal semantics.

## 1 Introduction

The derivation of provably correct static analyses from a formal specification of the semantics of a programming language is a long-standing challenge. The recent advances in the mechanisation of semantics has opened up novel perspectives for providing tool support for this task, thereby enabling the scaling of this approach to larger programming languages. This paper presents one such approach for mechanically constructing semantics-based program analysers from a formal description of the semantics of a programming language. We aim to provide methodologies which not only can prove the correctness of program abstractions but also lead to executable analysis techniques. Abstract Interpretation [4] has set out a methodology for defining an abstract semantics from an operational semantics and for proving a correctness relation between abstract and concrete semantics using Galois connections. The principle of abstract interpretation has been applied to a variety of semantic frameworks, including small-step and big-step (natural) operational semantics, and denotational semantics. An example of this methodology is to build an abstract semantics from a natural semantics [20]. Another example is Nielson's theory of abstract interpretation of two-level semantics [14] in which a semantic meta-language is equipped with binding-time annotations so that types and terms can be given a *static* and *dynamic* interpretation, leading to different but (logically) related interpretations.

In order for semantics-based program analysis to handle the complexity of today's programming languages, it is necessary to conceive a methodology that is built using some form of mechanised semantics. Examples of this include Verasco [8], a formally verified static analyser for the C programming language. It uses abstract interpretation techniques to perform value analyses, relational analyses... Verasco is written in Coq and the soundness of the analysis is guaranteed by a theorem: a program where the analysis does not raise an alarm is free of errors. Reasoning about program behaviours is possible as Verasco reuses the formalisation of the C semantics in Coq that was written for CompCert [11]. CompCert is a proved semantic preserving C compiler written in Coq.

Another example is the $\mathbb{K}$ [18] framework for writing semantics using rewriting rules. Rewriting rules make the formal definition of a semantics both flexible and relatively simple to write, and allows to mechanically derive objects from the semantics like an interpreter. However, this mechanization can be complex: $\mathbb{K}$-Java [2] is a formalization of Java in $\mathbb{K}$, with close to four hundred rewriting rules. It is unclear if it is possible to derive an analysis from a mechanization in $\mathbb{K}$.

G. Caltais and C. A. Mezzina (Eds): Combined Workshop on
Expressiveness in Concurrency and Structural Operational Semantics
(EXPRESS/SOS 2023).
EPTCS 387, 2023, pp. 97–113, doi:10.4204/EPTCS.387.8

The key idea that we will pursue in this paper is that an abstract interpreter for a semantic meta-language combined with language-specific abstractions for a particular property yield a correct-by-construction abstract interpreter for the specific language and property. We describe how to obtain a correct program analyser for a programming language from its *skeletal* semantics. Skeletal Semantics [1] is a proposal for machine-representable semantics of programming languages.

The skeletal semantics of a language $\mathscr{L}$ is a partial description of the semantics of $\mathscr{L}$. Typically, a skeletal semantics will contain definitions of the constructs of the language and functions of evaluation of these constructs. A skeletal semantics is written in the meta-language Skel [17], a minimalist functional language. It is a *meta language* to describe the semantics of *object languages*. Skel has several semantics, called *interpretations*, (small step, big step [10], abstract interpretation), giving different semantics for the object languages.

**Contributions**

- We propose new interpretations of the semantic meta-language Skel that integrates the notion of program point in a systematic way.
- We define an abstract interpretation for Skel. The abstract interpretation of Skel combined with language-specific abstractions define an analyzer for the object language.
- We prove that the abstract interpretation of Skel is a sound approximation of the big-step interpretation of Skel, provided that some small language-dependent properties hold.
- We implement a program which, given a Skeletal Semantics, generates an executable abstract interpreter, and we test it on toy languages. We define a basic value analyzer for a small imperative language. A Control Flow Analysis for a $\lambda$-calculus is also presented in the long version of this paper [7].

## 2 Skeletal Semantics

Skeletal Semantics offers a framework to mechanise semantics of programming languages [1]. It uses a minimalist, functional, and strongly typed semantic meta-language called Skel [17], whose syntax is presented in Figure 1. The actual semantics of a language described in Skel is expressed by providing a (meta-)interpretation of the Skel language itself. In this paper, we will present two such interpretations: a big-step (or concrete) semantics and an abstract interpretation.

We illustrate Skel through the definition of the skeletal semantics of a toy imperative language called While. A Skeletal Semantics is a formal description of a language and consists of *declarations*. We start with some type declarations (production $r_\tau$ in Figure 1).

```
type ident
type lit
type store
type int
```

```
type expr =
| Const lit
| Var ident
| Plus (expr, expr)
| Leq(expr, expr)
| Rand (lit, lit)
```

```
type stmt =
| Skip
| Assign (ident, expr)
| Seq (stmt, stmt)
| If (expr, stmt, stmt)
| While (expr, stmt)
```

For While, there are four *unspecified* types (identifiers, literals, stores, integers) and two *specified* types (expressions and statements). Unspecified types is an useful trait of Skel, their definitions are unconstrained and they can be instantiated depending on the semantics of the object language being defined.

$$
\begin{array}{rrcl}
\textsc{Term} & t & ::= & x \mid C\,t \mid (t,..,t) \mid \lambda p : \tau \to S \\
\textsc{Skeleton} & S & ::= & t \mid t_0\,t_1..t_n \mid \textbf{let}\ p = S\ \textbf{in}\ S \mid \textbf{branch}\ S \text{or} .. \text{or}\ S\ \textbf{end} \mid \\
 & & & \textbf{match}\ t\ \textbf{with}\ p \to S..p \to S\ \textbf{end} \\
\textsc{Pattern} & p & ::= & x \mid \_ \mid C\ p \mid (p,..,p) \\
\textsc{Type} & \tau & ::= & b \mid \tau \to \tau \mid (\tau,..,\tau) \\
\textsc{Term decl} & r_t & ::= & \textbf{val}\ x : \tau \mid \textbf{val}\ x : \tau = t \\
\textsc{Type decl} & r_\tau & ::= & \textbf{type}\ b \mid \textbf{type}\ b = \texttt{"|"}\ C_1\ \tau_1 .. \texttt{"|"}\ C_n\ \tau_n \\
\textsc{Skeletal Semantics} & \mathscr{S} & ::= & (r_t | r_\tau)^*
\end{array}
$$

Figure 1: The Syntax of Skeletal Semantics

The specification of the integer type can be different for a big-step semantics or for an abstract interpretation. The `expr` and `stmt` types define expressions and statements of While programs. An expression can be a constant, a variable, an addition, a comparison, or a random integer. A statement can be a skip (an instruction that does nothing), an assignment, a sequence, a condition, or a loop. In addition to these declared types, one may build arrow types and tuple types.

We now turn to Skel's *term declarations* (production $r_t$ of Figure 1), which may also be unspecified or specified. Unspecified terms are typically used for operations on values of unspecified types. For our While language, they are as follows.

```
val litToInt : lit → int                    val isZero: int → ()
val add : (int,  int) → int                 val isNotZero: int → ()
val lt: (int,  int) → int                   val read : (ident, store) → int
val rand : (lit, lit) → int                 val write : (ident, store, int) → store
```

The types for `isZero` and `isNotZero` may be surprising. These partial functions act as filters when used in branches, as detailed below.

Specified terms, on the other hand, are signatures associated with a *term*. A term is either a skeletal variable, a constructor applied to a term, a tuple, or an abstraction. The body of an abstraction is a skeleton, described below. Consider the declaration of term `eval_expr`.

```
val eval_expr ((s, e): (store, expr)): int =
  match e with
  | Const i → litToInt i                    | Leq (e1, e2) →
  | Var x → read (x, s)                        let v1 = eval_expr (s, e1) in
  | Plus (e1, e2) →                            let v2 = eval_expr (s, e2) in
     let v1 = eval_expr (s, e1) in             lt (v1, v2)
     let v2 = eval_expr (s, e2) in          | Rand (i1, i2) → rand (i1, i2)
     add (v1, v2)                           end
```

The first line is syntactic sugar for

```
val eval_expr: (store, expr) -> int = λ (s, e): (store, expr) ->
```

where the remainder of the description is the body of the abstraction. This body is a skeleton. A skeleton may be a term, an n-ary application, a let binding, a branching (detailed below), or a match. Here the

skeleton is a match, distinguishing between the different expressions which may be evaluated. For a constant expression, we call the unspecified term `litToInt` to convert the literal to an integer. For a variable, we read its value in the store. For an addition, we sequence the recursive evaluation of each subterm using a let binding, and we then apply the unspecified `add` term to perform the actual addition. Note that specified term and type declarations are all mutually recursive. The rest of the code does not use any additional feature.

We now turn to the second specified term declaration, to evaluate statements.

```
val eval_stmt ((s, t): (store, stmt)): store  =
  match t with
  | Skip → s
  | Assign (x, e) →
     let w = eval_expr (s, e) in
     write (x, s, w)
  | Seq (t1, t2) →
     let s' = eval_stmt (s, t1) in
     eval_stmt (s', t2)
  | If (cond, true, false) →
     let i = eval_expr (s, cond) in
     branch
       let () = isNotZero i in
       eval_stmt (s, true)
     or
       let () = isZero i in
       eval_stmt (s, false)
     end
  | While (cond, t') →
     let i = eval_expr (s, cond) in
     branch
       let () = isNotZero i in
       let s' = eval_stmt (s, t') in
       eval_stmt (s', t)
     or
       let () = isZero i in s
     end
end
```

The code for the conditional and the loop illustrates the last feature of the language, branching. Branches are introduced with the `branch` keyword and are separated with the `or` keyword. They correspond to a form of a non-deterministic choice. Intuitively, in a big-step interpretation, any branch that succeeds may be taken. Branches may fail if a pattern matching in a let binding fails, or if the application of a term fails. For instance, the instantiation of the term `isNotZero` will not be defined on 0, making the whole branch fail when given 0 as argument. This is how we decide which branch to execute next for the conditional, and whether to loop in the `While` case.

## 3 Big-step Semantics of Skel

We give meaning to a Skeletal Semantics by providing *interpretations* of Skel. We first define the concrete, big-step semantics of Skel. Let $\mathscr{S}$ be an arbitrary Skeletal Semantics. We write $\text{Funs}(\mathscr{S})$ for the set of pairs $(\Gamma, \lambda\, p : \tau_1 \to S_0)$ such that $\lambda\, p : \tau_1 \to S_0$ appears in Skeletal Semantics $\mathscr{S}$. The typing environment $\Gamma$ gives types to the free variables of $\lambda\, p : \tau_1 \to S_0$. Full formal details are available in [7].

### 3.1 From Types to Concrete Values

The definitions of the sets of semantic values are presented on Figure 2a. They are defined by induction on the type. For each type $\tau$, we write $V(\tau)$ the set of values of type $\tau$.

A value with tuple type is a tuple of concrete values. A value of a specified type is a constructor applied to a value. A value with arrow type is a function that can either be a *named closure* or an *anonymous closure*. The set of named closure $\text{NC}(\tau_1 \to \tau_2)$ and the set of anonymous closures $\text{AC}(\tau_1 \to \tau_2)$ are defined on Figure 2b. A named closure denotes a function that is specified in the Skeletal Semantics

$$V(\tau_1 \times .. \times \tau_n) = V(\tau_1) \times .. \times V(\tau_n)$$
$$V(\tau_2) = \{\, C\ v \mid C : (\tau_1, \tau_2) \wedge v \in V(\tau_1) \,\}$$
$$V(\tau_1 \to \tau_2) = \mathrm{NC}(\tau_1 \to \tau_2) \cup \mathrm{AC}(\tau_1 \to \tau_2)$$

(a) Concrete values associated to each type

$$\mathrm{NC}(\tau_1 \to \tau_2) = \left\{ (f,n) \;\middle|\; \begin{array}{c} \textbf{val}\ f : \tau_1 \to \tau_2 [= t] \in \mathscr{S} \\ \mathrm{arity}(f) = n \end{array} \right\}$$
$$\mathrm{AC}(\tau_1 \to \tau_2) = \left\{ (\Gamma, p, S, E) \;\middle|\; \begin{array}{c} (\Gamma, \lambda p : \tau_1 \to S) \in \mathrm{Funs}(\mathscr{S}) \\ \Gamma \vdash E \\ \Gamma + p \leftarrow \tau_1 \vdash S : \tau_2 \end{array} \right\}$$

(b) Named Closures and Anonymous Closures

Figure 2: Definition of Concrete Values

$\mathscr{S}$, it is a pair of the name of the function and its arity. An anonymous closure is a tuple of a typing environment $\Gamma$, a pattern $p$ to bind the argument upon application, a skeleton $S$ which is the body of the function, and an environment $E$ captured at the creation of the closure. An environment is a partial function mapping skeletal variable to concrete values. It is said to be consistent with typing environment $\Gamma$, written $\Gamma \vdash E$, if they have the same domain and if, for every $x \in \mathrm{dom}(\Gamma)$, we have $E(x) \in V^\sharp(\Gamma(x))$.

The unspecified types of a skeletal semantics must be instantiated to obtain an interpretation. In the case of While, the unspecified types are `ident`, `lit`, `int`, and `store`. They are instantiated as follows.

$$V(\texttt{store}) = \{\, s \mid s \in \mathscr{X} \hookrightarrow \mathbb{Z} \,\} \qquad V(\texttt{ident}) = \{\, x \mid x \in \mathscr{X} \,\} \qquad \text{with } \mathscr{X} = \{x, y, z, ..\}$$
$$V(\texttt{lit}) = \{\, l \mid l \in \mathbb{Z} \,\} \qquad V(\texttt{int}) = \{\, i \mid i \in \mathbb{Z} \,\}$$

Identifiers are taken from a countable set $\mathscr{X}$, literals and integers are relative integers, and stores are partial maps from identifiers to integers.

## 3.2 Interpretation of Unspecified Terms

In the following, we write $\mathrm{na}(\tau)$ when $\tau$ is not an arrow type. Take an unspecified term **val** $t : \tau_1 \to .. \to \tau_n \to \tau$ such that $\mathrm{na}(\tau)$, then an instantiation of $t$, written $[\![t]\!]$, is a function such that $[\![t]\!] \in (V(\tau_1) \times .. \times V(\tau_n)) \to \mathscr{P}_{fin}(V(\tau))$, where $\mathscr{P}_{fin}(X)$ is the set of finite subsets of $X$. In particular, if **val** $t : \tau$ and $\mathrm{na}(\tau)$, then $[\![t]\!] \subseteq V(\tau)$. Allowing the specification of a term to be a function returning a set is useful to model non-determinism.

We instantiate the unspecified functions of our While language. The expression $(b)$ **?** $e_1$ **:** $e_2$ evaluates

$$\frac{E, S_1 \Downarrow v \qquad \vdash E + p \leftarrow v \rightsquigarrow E' \qquad E', S_2 \Downarrow w}{E, \text{let } p = S_1 \text{ in } S_2 \Downarrow w}\ \textsc{LetIn} \qquad \frac{E, S_i \Downarrow v}{E, (S_1, .., S_n) \Downarrow v}\ \textsc{Branch}$$

Figure 3: Examples of Rules of the Big-Step Semantics

$$\frac{}{\vdash E + \_ \leftarrow v \rightsquigarrow E}\ \textsc{wild} \qquad \frac{}{\vdash E + x \leftarrow v \rightsquigarrow \{x \mapsto v\} E}\ \textsc{var} \qquad \frac{\vdash E + p \leftarrow v \rightsquigarrow E'}{\vdash E + C\,p \leftarrow C\,v \rightsquigarrow E'}\ \textsc{const}$$

$$\frac{\vdash E + p_1 \leftarrow v_1 \rightsquigarrow E_2 \qquad .. \qquad \vdash E_n + p_n \leftarrow v_n \rightsquigarrow E'}{\vdash E + (p_1, .., p_n) \leftarrow (v_1, .., v_n) \rightsquigarrow E'}\ \textsc{tuple}$$

Figure 4: Rule of Extension of Environment using Pattern Matching

to $e_1$ is the condition $b$ is true. Otherwise, it evaluates to $e_2$.

$$\begin{aligned}
[\![litToInt]\!](n) &= \{n\} & [\![add]\!](n_1, n_2) &= \{n_1 + n_2\} \\
[\![lt]\!](n_1, n_2) &= (n_1 < n_2)\ \mathbf{?}\ \{1\}\ \mathbf{:}\ \{0\} & [\![rand]\!](n_1, n_2) &= \{n \mid n_1 \leq n \leq n_2\} \\
[\![isZero]\!](n) &= (n = 0)\ \mathbf{?}\ \{()\}\ \mathbf{:}\ \{\} & [\![isNotZero]\!](n) &= (n \neq 0)\ \mathbf{?}\ \{()\}\ \mathbf{:}\ \{\} \\
[\![read]\!](x, s) &= \{s(x)\} & [\![write]\!](x, s, n) &= \{s\{x \mapsto n\}\}
\end{aligned}$$

The `rand` instantiation returns a set of values to capture the non-determinism of the instruction. The `isZero` function is defined only on input 0, whereas `isNotZero` is defined for all inputs except 0.

### 3.3 Big-step Semantics

We briefly present the big-step semantics of Skel: $E, S \Downarrow v$ is a relation from a skeletal environment $E$, mapping skeletal variables to values, and a skeleton $S$ to a value $v$. The relation is defined by induction on $S$ and is very similar to the natural semantics of $\lambda$-calculus with environment. We focus on two of the most important rules on Figure 3. The LetIn rule evaluates a let-binding by first evaluating $S_1$, next binding the result to the pattern in the current environment, then finally evaluating $S_2$ in the extended environment. The Branch rule describes how to evaluate a branching: any branch that successfully reduces to a value may be taken. Finally, the pattern matching rules for environment extension are given in Figure 4. The whole set of rules is given in [7].

The big-step semantics of a branching explains the types of some unspecified terms seen earlier. The partiality of the instantiations of `isZero` and `isNotZero` functions are used in the semantics of While to prevent some branches to be taken. They act as filters: if a branch does not have a derivation because its filter is undefined on the input, the alternative is to take another branch.

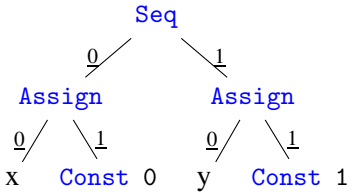
## 4 Big-step Semantics with Program Points

In this section, we introduce our first contribution, which is the integration of the notion of *program point* into the framework of Skeletal Semantics. A program point maps-to a precise fragment of a given

program. They play an important role in semantics-based program analysis, to indicate places where information about the execution is collected. Program points are essential to abstract interpretation as an abstract interpretation usually computes an abstraction of the state of the execution of the analysed program for each program point. Our formalisation of program points for Skeletal Semantics is modular and works for the big-step semantics of Skel, but also for the abstract interpretation of Skel, presented in Section 5.

In Skel, programs are values of an algebraic data type (ADT), such as `stmt` or `expr` in the While example. For instance, the skeletal term `Seq(Assign(x, Const 0), Assign(y, Const 1))` is a While **program** of type `stmt`. A program point is a path in the ADT of the program, encoded as a list of integers (underlined to distinguish them from natural numbers).

For example, $\varepsilon$ is the empty path, it corresponds to the whole program. The path $\underline{01}$ corresponds to `Const 0`. The set of program points is thus $\texttt{ppt} = \mathbb{N}^*$.




Let **prg** be a term of an ADT and pp a program point. We note **prg**@ pp the subterm of **prg** at program point pp. Formally, it is defined as follows.

$$v@\underline{\varepsilon} = v \qquad \texttt{C}(v_0,..,v_{n-1})@(\underline{i}\,\text{pp}) = v_i@\,\text{pp} \quad \text{when } 0 \le i \le n-1$$

## 4.1 Building Values with Program Points

Our approach is to replace the values that correspond to programs with program points. These program points correspond to a sub-program of a main program that is a parameter of the interpretation. The values that ought to be replaced by program points should be values representing fragments of the program being executed. Therefore, we call $\mathscr{T}$ the set of *program types*, i.e., types representing programs. For instance, for the While language, $\mathscr{T} = \{\texttt{stmt}, \texttt{expr}\}$. Moreover, the interpretation is parametrised by a program **prg**, which is a value of a type $\tau \in \mathscr{T}$. For the While language, that could be **prg** = `Seq(Assign(x, Const 0), Assign(y, Const 1))`. Therefore, the rules to build values are unchanged except for the values with program types. Values with program types are defined by the following equation, where $V(\tau)$ is defined in Figure 2a.

$$\tau \in \mathscr{T} \implies V^{\texttt{ppt}}_{\textbf{prg}}(\tau) = \{\, \text{pp} \in \texttt{ppt} \mid \textbf{prg}@\,\text{pp} \in V(\tau) \,\}$$

Therefore in our example, $\varepsilon$ is now a value of type `stmt` denoting the value **prg**, and $\underline{0}$ denotes the value `Assign(x, Const 0)`.

For each unspecified term $x$, we assume given an interpretation $[\![x]\!]^{\texttt{ppt}}$, which is identical to the concrete interpretation $[\![x]\!]$ where program terms are replaced by program points. The full definition of this interpretation can be found in the long version of this paper [7].

## 4.2 Pattern-matching of Program Points

Replacing some values with program points does not change the interpretation of Skel, except when matching a program point with a pattern. Indeed, a program point pp corresponds to the sub-program **prg**@ pp if it exists, and it might be matched against a pattern `C` $p$. To handle this case, the program point is *unfolded*, meaning the constructor at **prg**@ pp is revealed, and the parameters of the constructor are replaced with program points if their type is a program type. To give an example, given **prg** as before, unfolding $\varepsilon$ gives `Seq`$(\underline{0},\underline{1})$: the constructor is revealed and the parameters are program points because

they both have type `stmt` $\in \mathscr{T}$. On the other hand, unfolding $\underline{00}$ directly returns `x`, as identifiers are not program types in this example. This unfolding mechanism is added to pattern matching via the following rule:

$$\frac{\begin{array}{c}\mathbf{prg}@\,\mathrm{pp} = \mathtt{C}(v'_0,..,v'_{n-1}) \qquad \mathtt{C} : (\tau_0 \times .. \times \tau_{n-1}, \tau) \\ v_j = \mathbf{if}\ \tau_j \in \mathscr{T}\ \mathbf{then}\ \mathrm{pp}\ \underline{j}\ \mathbf{else}\ v'_j \qquad \mathscr{T}, \mathbf{prg} \vdash E + p \leftarrow (v_0,..,v_{n-1}) \rightsquigarrow E'\end{array}}{\mathscr{T}, \mathbf{prg} \vdash E + \mathtt{C}\,p \leftarrow \mathrm{pp} \rightsquigarrow E'}\ \textsc{unfold}$$

Note that the pattern matching is now parametrised with $\mathscr{T}$ and **prg**. To perform the pattern-matching of pp with `C` $p$, the value **prg**@ pp must have constructor `C` at the root. The parameters that have a program type are replaced by their program point and the pattern-matching is performed recursively.

# 5 Abstract Interpretation of Skel

We present our main contribution in this paper, which is the definition of an abstract interpreter of Skel that is sound with respect to the big-step semantics of Section 4. This abstract interpreter will serve as the foundations of a methodology for building abstract interpreters for object languages from their skeletal semantics. In this methodology, several ingredients must be provided to generate such an abstract interpreter.

- An abstract instantiation of unspecified types to sets of abstract values, each of which comes with a concretisation function, a partial order, and an abstract union operator. Our framework automatically extends these definitions to all types.
- A *state of the abstract interpretation* (AI-state in the following) used to carry additional information during the abstract evaluation. For instance, in our While language, the AI-state records the current approximation of the store for each program point.
- The user may give functions to update the AI-state at the start and end of a call to a specified function. This is typically used for evaluation functions, such as `eval_stmt`, to record information right before and right after executing a sub-program.
- An instantiation of the unspecified terms, based on the definition of the abstract instantiation of types and the AI-state.

Our framework provides an abstract meta-semantics of Skel that threads the AI-state through the evaluation, including calls to unspecified terms. As the foundational correctness property of the methodology, we prove that if the abstract instantiation of types and terms provided by the user satisfy some correctness criteria, then the whole abstract interpreter that is generated is also correct.

## 5.1 Abstract Values

Abstract values are built similarly to concrete ones, based on their types, and we write $V^\sharp(\tau)$ for the set of abstract values of type $\tau$. We first define the sets of abstract (named) closures at the top of Figure 5. Abstract named closures are identical to concrete named closures: they are pairs of a name of a function defined in the skeletal semantics and its arity. Abstract closures consist of a typing environment, a pattern, a skeleton, and an abstract environment that is consistent with the typing environment. An abstract environment $E^\sharp$ is a mapping from skeletal variables to abstract values. It is said to be consistent with typing environment $\Gamma$, written $\Gamma \vdash E^\sharp$, if they have the same domain and if, for every $x \in \mathrm{dom}(\Gamma)$, we have $E^\sharp(x) \in V^\sharp(\Gamma(x))$.

$$\mathrm{NC}(\tau_1 \to \tau_2) = \left\{ (f,n) \;\middle|\; \begin{array}{c} \textbf{val}\ f : \tau_1 \to \tau_2[=t] \in \mathscr{S} \\ \mathrm{arity}(f) = n \end{array} \right\}$$

$$\mathrm{AC}(\tau_1 \to \tau_2) = \left\{ (\Gamma, p, S, E^\sharp) \;\middle|\; \begin{array}{c} (\Gamma, \lambda p : \tau_1 \to S) \in \mathrm{Funs}(\mathscr{S}) \\ \Gamma \vdash E^\sharp \\ \Gamma + p \leftarrow \tau_1 \vdash S : \tau_2 \end{array} \right\}$$

$$V^\sharp(\tau_1 \times .. \times \tau_n) = \mathscr{P}_{fin}(V^{\sharp\star}(\tau_1) \times .. \times V^{\sharp\star}(\tau_n))$$

$$V^\sharp(\tau_2) = \{C\ v^\sharp \mid C : (\tau_1, \tau_2) \wedge v^\sharp \in V^{\sharp\star}(\tau_1)\} \cup \{\bot_{\tau_2}, \top_{\tau_2}\}$$

$$V^\sharp(\tau_1 \to \tau_2) = \mathscr{P}(\mathrm{NC}(\tau_1 \to \tau_2)) \cup \mathscr{P}(\mathrm{AC}(\tau_1 \to \tau_2))$$

Figure 5: Abstract Values for Specified Types

We assume that a partial order on $V^\sharp(\tau)$ is provided for each unspecified type $\tau$, and that it includes a smallest value, denoted by $\bot_\tau$, and a largest value, denoted by $\top_\tau$. In the case of While, we instantiate `ident` with the flat lattice of $\mathscr{X}$, `lit` with the flat lattice of integers, `int` with closed intervals of $\mathbb{Z} \cup \{-\infty, +\infty\}$, and `store` with a partial mapping from identifiers to non-empty intervals.

We define abstract values for specified types in Figure 5, writing $V^{\sharp\star}(\tau)$ for $V^\sharp(\tau) \setminus \{\bot_\tau\}$. Abstract tuples are finite sets of tuples of (non-bottom) abstract values, with $\bot_{\tau_1 \times .. \times \tau_n}$ being the empty set. We use sets to retain some precision in the analysis. Abstract values of an algebraic data type are simply constructors applied to an abstract value of the correct type. Finally, functional abstract values are sets of abstract closures of this type, with $\bot_{\tau_1 \to \tau_2}$ being the empty set.

The AI-state $\mathscr{A}$ contains information collected throughout the abstract interpretation. It is dependent on the analysis and the language, and therefore must be provided, similarly to unspecified values. Moreover, a partial order and a union must be given for abstract states. In the case of while, the AI-state records information about the abstract store before (`In`) and after (`Out`) every program point. We write `Pos` for either `In` or `Out`. We then define $\mathscr{A}$ as a mapping from program points and a `Pos` to abstract while stores. We have $\mathscr{A}_1 \sqsubseteq^\sharp \mathscr{A}_2$ if $\mathrm{dom}(\mathscr{A}_1) \subseteq \mathrm{dom}(\mathscr{A}_2)$ and for any $(\mathrm{pp}, \mathtt{Pos}) \in \mathrm{dom}(\mathscr{A}_1)$, we have $\mathscr{A}_1(\mathrm{pp}, \mathtt{Pos}) \sqsubseteq^\sharp_{store} \mathscr{A}_2(\mathrm{pp}, \mathtt{Pos})$. We define $\mathscr{A}_1 \sqcup^\sharp \mathscr{A}_2$ as the mapping from $\mathrm{dom}(\mathscr{A}_1) \cup \mathrm{dom}(\mathscr{A}_2)$ that relates $(\mathrm{pp}, \mathtt{Pos})$ to $\mathscr{A}_1(\mathrm{pp}, \mathtt{Pos}) \sqcup^\sharp_{int} \mathscr{A}_2(\mathrm{pp}, \mathtt{Pos})$.

## 5.2 Operations on Abstract Values

Each abstract domain has a partial order and an associated join operator. In addition, a concretisation function that returns a set of concrete values defines the meaning of each abstract value. All of these functions are indexed by types (or type environments when they deal with environments). We assume they are provided for non-specified types, and show in this section how to extend them to all types.

A concretisation function for type $\tau$ maps an abstraction state and an abstract value in $V^\sharp(\tau)$ to $\mathscr{P}(V(\tau))$, a set of concrete values. We also define a function of concretisation $\gamma_\Gamma$ which maps abstract

skeletal environments to sets of concrete skeletal environments.

$$\gamma_{\tau_1\times..\times\tau_n}(\mathscr{A},t^\sharp) = \bigcup_{(v_1^\sharp,..,v_n^\sharp)\in t^\sharp} \gamma_{\tau_1}(\mathscr{A},v_1^\sharp)\times..\times\gamma_{\tau_n}(\mathscr{A},v_n^\sharp)$$
$$\gamma_{\tau_2}(\mathscr{A},C\,v^\sharp) = \left\{ C\,v \mid C:(\tau_1,\tau_2),\, v\in\gamma_{\tau_2}(\mathscr{A},v^\sharp) \right\}$$
$$\gamma_{\tau_1\to\tau_2}(\mathscr{A},F) = \{\,(f,n)\mid(f,n)\in F\,\}\cup\left\{ (\Gamma,p,S,E) \,\middle|\, (\Gamma,p,S,E^\sharp)\in F \wedge E\in\gamma_\Gamma(\mathscr{A},E^\sharp) \right\}$$
$$\gamma_\Gamma(\mathscr{A},E^\sharp) = \left\{ E \,\middle|\, \Gamma\vdash E\wedge\Gamma\vdash E^\sharp\wedge\forall x\in\mathrm{dom}(\Gamma), E(x)\in\gamma_{\Gamma(x)}(\mathscr{A},E^\sharp(x)) \right\}$$
$$\gamma(\mathscr{A},\bot_\tau)=\emptyset \qquad \gamma(\mathscr{A},\top_\tau)=V^\sharp(\tau)$$

In the case of While, the concretisation function for `ident` and `lit` are immediate as they are flat lattices. The concretisation function for an interval $i$ is the set of integers it contains: $\gamma_{\mathtt{int}}(i)=\{\,n\mid n\in i\,\}$. Finally, the concretisation of an abstract store $\sigma^\sharp$ is

$$\gamma_{\mathtt{store}}(\sigma^\sharp) = \left\{ \sigma \,\middle|\, \mathrm{dom}(\sigma)=\mathrm{dom}(\sigma^\sharp)\wedge\forall x\in\mathrm{dom}(\sigma), \sigma(x)\in\gamma_{\mathtt{int}}(\sigma^\sharp(x)) \right\}$$

To compare abstract values, we define partial orders that are relations, but we call them functions as they can be viewed as boolean functions. For every unspecified type $\tau$, we assume a comparison function $\sqsubseteq^\sharp_\tau$ which is a partial order between abstract values. It must satisfy the following property: for any value $v^\sharp\in V^\sharp(\tau)$, we have $\bot_\tau\sqsubseteq^\sharp_\tau v^\sharp$ and $v^\sharp\sqsubseteq^\sharp_\tau\top_\tau$. For every other type, the comparison function is the smallest partial order that satisfies the following equations.

$$C\,v^\sharp\sqsubseteq^\sharp_{\tau_a} C\,w^\sharp \iff v^\sharp\sqsubseteq^\sharp_\tau w^\sharp \text{ with } C:(\tau,\tau_a)$$
$$v^\sharp\sqsubseteq^\sharp_{\tau_1\times..\times\tau_n} w^\sharp \iff \forall(v_1^\sharp,..,v_n^\sharp)\in v^\sharp, \exists(w_1^\sharp,..,w_n^\sharp)\in w^\sharp \text{ such that } \forall i\in[1..n],\, v_i^\sharp\sqsubseteq^\sharp_{\tau_i} w_i^\sharp$$
$$F_1\sqsubseteq^\sharp_{\tau_1\to\tau_2} F_2 \iff \begin{cases} (f,n)\in F_1 \implies (f,n)\in F_2 \\ (\Gamma,p,S,E_1^\sharp)\in F_1 \implies \exists(\Gamma,p,S,E_2^\sharp)\in F_2, E_1^\sharp\sqsubseteq^\sharp_\Gamma E_2^\sharp \end{cases}$$
$$E_1^\sharp\sqsubseteq^\sharp_\Gamma E_2^\sharp \iff \Gamma\vdash E_1^\sharp\wedge\Gamma\vdash E_2^\sharp\wedge\forall x\in\mathrm{dom}(E_1^\sharp), E_1^\sharp(x)\sqsubseteq^\sharp_{\Gamma(x)} E_2^\sharp(x)$$
$$v^\sharp\sqsubseteq^\sharp_\tau\top_\tau \qquad \bot_\tau\sqsubseteq^\sharp_\tau v^\sharp$$

Most rules are straightforward. To compare two functions, all named closures of the left function must be in the right function. Moreover, for all closures in the left function, there must be a closure in the right function with the same pattern and skeleton, but with a bigger abstract environment. Abstract environments are compared using point-wise lifting. For our While language, we have $i_1\sqsubseteq^\sharp_{\mathtt{int}} i_2$ if the interval $i_1$ is included in $i_2$, and $\sigma_1^\sharp\sqsubseteq^\sharp_{\mathtt{store}}\sigma_2^\sharp$ if for all $x$ in $\mathrm{dom}(\sigma_1^\sharp)$, we have $\sigma_1^\sharp(x)\sqsubseteq^\sharp_{\mathtt{int}}\sigma_2^\sharp(x)$.

**Definition 1** *A concretion function $\gamma_\tau$ is* monotonic *iff for any $v_1^\sharp\sqsubseteq^\sharp_\tau v_2^\sharp$ and $\mathscr{A}_1\sqsubseteq^\sharp\mathscr{A}_2$. we have $\gamma_\tau(\mathscr{A}_1,v_1^\sharp)\subseteq\gamma_\tau(\mathscr{A}_2,v_2^\sharp)$.*

**Lemma 1** *$\gamma_{\mathtt{ident}}$, $\gamma_{\mathtt{lit}}$, $\gamma_{\mathtt{int}}$ and $\gamma_{\mathtt{store}}$ are monotonic.*

For each type, an upper bound (or join) is defined. For every non-specified type $\tau$, we assume an upper bound $\sqcup^\sharp{}_\tau$. The definition of $\sqcup^\sharp{}_{\mathtt{ident}}$ and $\sqcup^\sharp{}_{\mathtt{lit}}$ have the usual definition for flat lattices. $\sqcup^\sharp{}_{\mathtt{int}}$ is the convex hull of two intervals and $\sqcup^\sharp{}_{\mathtt{store}}$ is the usual point-wise lifting of the abstract union of

integers. Moreover, we note $\nabla^\sharp_{\texttt{int}}$ the widening on intervals (define below) and $\nabla^\sharp_{\texttt{store}}$ the point-wise lifting of the widening of intervals.

$$[n_1,n_2]\nabla^\sharp_{\texttt{int}}[m_1,m_2] = \left[\begin{cases} n_1 & \text{if } n_1 \leq m_1 \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} n_2 & \text{if } m_2 \leq n_2 \\ +\infty & \text{otherwise} \end{cases}\right]$$

We extend it for every other type.

$$(C\ v^\sharp)\sqcup^\sharp_{\tau_2}(C\ w^\sharp) = C\ (v^\sharp\sqcup^\sharp_{\tau_1} w^\sharp) \text{ with } C:(\tau_1,\tau_2) \qquad E_1^\sharp\sqcup^\sharp_\Gamma E_2^\sharp = \left\{ x \in \mathrm{dom}(\Gamma) \mapsto E_1^\sharp(x)\sqcup^\sharp_{\Gamma(x)} E_2^\sharp(x) \right\}$$

$$(C\ v^\sharp)\sqcup^\sharp_{\tau_2}(D\ w^\sharp) = \top_{\tau_2} \text{ with } C:(\tau_1,\tau_2) \wedge D:(\tau_1',\tau_2) \qquad v^\sharp\sqcup^\sharp_\tau \top_\tau = \top_\tau \sqcup^\sharp_\tau v^\sharp = \top_\tau$$

$$v^\sharp\sqcup^\sharp_{\tau_1\times..\times\tau_n} w^\sharp = v^\sharp \cup w^\sharp \qquad v^\sharp\sqcup^\sharp_\tau \bot_\tau = \bot_\tau\sqcup^\sharp_\tau v^\sharp = v^\sharp$$

$$F_1\sqcup^\sharp_{\tau_1\to\tau_2} F_2 = F_1 \cup F_2$$

Joining two algebraic values with the same constructor is joining their parameters, and joining algebraic values with different constructors yields top. The join of abstract tuples or abstract functions is their union. Joining abstract environments is done by point-wise lifting. For each type, top is an absorbing element, and bottom is the neutral element.

**Lemma 2** *$\sqsubseteq^\sharp_{\texttt{ident}}$, $\sqsubseteq^\sharp_{\texttt{lit}}$, $\sqsubseteq^\sharp_{\texttt{int}}$, $\sqsubseteq^\sharp_{\texttt{store}}$ are orders. $\sqcup^\sharp_{\texttt{ident}}$, $\sqcup^\sharp_{\texttt{lit}}$, $\sqcup^\sharp_{\texttt{int}}$, $\sqcup^\sharp_{\texttt{store}}$ give an upper bound.*

**Lemma 3** *If for all unspecified types $\tau_u$, $\gamma_{\tau_u}$ is monotonic, then for all $\tau$, $\gamma_\tau$ is also monotonic.*

**Lemma 4** *If for every unspecified type $\tau_u$, $\sqsubseteq^\sharp_{\tau_u}$ is an order and $\sqcup^\sharp_{\tau_u}$ gives an upper bound, then for all $\tau$, $\sqsubseteq^\sharp_\tau$ is an order and $\sqcup^\sharp_\tau$ gives an upper bound.*

Finally, we give an abstract specification of unspecified terms. As an illustration, here are a few specifications from our running example.

$$[\![litToInt]\!]^\sharp(n) = [n,n] \qquad [\![add]\!]^\sharp([n_1,n_2],[m_1,m_2]) = [n_1+m_1,n_2+m_2]$$

$$[\![read]\!]^\sharp(x,s^\sharp) = s^\sharp(x) \qquad [\![write]\!]^\sharp(x,s,[n_1,n_2]) = s^\sharp\{x\mapsto[n_1,n_2]\}$$

**Definition 2** *Let $x$ be an unspecified term of type $\tau$, such that* $\mathrm{na}(\tau)$. *We say that $[\![x]\!]^\sharp$ is a **sound approximation** of $[\![x]\!]^{ppt}$ if and only if:*

$$\forall \mathscr{A},\ [\![x]\!]^{ppt} \subseteq \gamma(\mathscr{A},[\![x]\!]^\sharp)$$

**Definition 3** *Let $f$ be an unspecified term of type $\tau_1\to..\to\tau_n\to\tau$ where* $\mathrm{na}(\tau)$. *We say that $[\![f]\!]^\sharp$ is a **sound approximation** of $[\![f]\!]^{ppt}$ iff $\forall v_i \in V^{ppt}_{prg}(\tau_i)$, $\forall v_i^\sharp \in V^\sharp(\tau_i)$, and for all abstract state $\mathscr{A}$, if*

$$\left.\begin{array}{r} v_i \in \gamma_{\tau_i}(\mathscr{A},v_i^\sharp) \\ [\![f]\!]^\sharp(\mathscr{A},v_1^\sharp,..,v_n^\sharp) = \mathscr{A}',w^\sharp \end{array}\right\} \implies [\![f]\!]^{ppt}(v_1,..,v_n) \subseteq \gamma_\tau(\mathscr{A}',w^\sharp)$$

**Lemma 5** *The abstract instantiations of the unspecified terms for While are sound approximation of the concrete instantiations of the unspecified terms.*

$$\frac{E^\sharp(x) = v^\sharp}{E^\sharp, x \Downarrow v^\sharp}\ \textsc{Var} \qquad \frac{\textbf{val}\ f : \tau_1 \to .. \to \tau_n \to \tau [= t] \in \mathscr{S} \qquad \mathrm{na}(\tau)}{E^\sharp, f \Downarrow \{ (f, n) \}}\ \textsc{TermClos}$$

$$\frac{\textbf{val}\ x : \tau = t \in \mathscr{S} \qquad \emptyset, t \Downarrow v^\sharp \qquad \mathrm{na}(\tau)}{E^\sharp, x \Downarrow v^\sharp}\ \textsc{TermSpec} \qquad \frac{\textbf{val}\ x : \tau \in \mathscr{S} \qquad \mathrm{na}(\tau)}{E^\sharp, x \Downarrow [\![x]\!]^\sharp}\ \textsc{TermUnspec}$$

$$\frac{E^\sharp, t \Downarrow v^\sharp}{E^\sharp, \mathrm{C}\, t \Downarrow \mathrm{C}\, v^\sharp}\ \textsc{Const} \qquad \frac{E^\sharp, t_1 \Downarrow v_1^\sharp \qquad .. \qquad E^\sharp, t_n \Downarrow v_n^\sharp}{E^\sharp, (t_1, .., t_n) \Downarrow \left\{ (v_1^\sharp, .., v_n^\sharp) \right\}}\ \textsc{Tuple}$$

$$\frac{}{\pi, E^\sharp, \lambda\, p : \tau \cdot S \Downarrow^\sharp \left\{ (p, S, E^\sharp) \right\}}\ \textsc{Clos} \qquad \frac{\pi, \mathscr{A}, E^\sharp, S_i \Downarrow^\sharp v_i^\sharp, \mathscr{A}_i}{\pi, \mathscr{A}, E^\sharp, (S_1 .. S_n) \Downarrow^\sharp \sqcup^\sharp v_i^\sharp, \sqcup^\sharp \mathscr{A}_i}\ \textsc{Branch}$$

$$\frac{\pi, \mathscr{A}, E^\sharp, S_1 \Downarrow^\sharp v^\sharp, \mathscr{A}' \qquad \mathscr{T}, \textbf{prg} \vdash \left\{ E^\sharp \right\} + p \leftarrow v^\sharp \rightsquigarrow \left\{ E_1^\sharp, .., E_n^\sharp \right\} \qquad \pi, \mathscr{A}', E_i^\sharp, S_2 \Downarrow^\sharp w_i^\sharp, \mathscr{A}_i}{\pi, \mathscr{A}, E^\sharp, \mathrm{let}\, p = S_1 \,\mathrm{in}\, S_2 \Downarrow^\sharp \sqcup^\sharp w_i^\sharp, \sqcup^\sharp \mathscr{A}_i}\ \textsc{LetIn}$$

$$\frac{E^\sharp, t_i \Downarrow v_i^\sharp \qquad \pi, \mathscr{A}, v_0^\sharp\ v_1^\sharp .. v_n^\sharp \Downarrow_{app} v^\sharp, \mathscr{A}'}{\pi, \mathscr{A}, E^\sharp, t_0\ t_1 .. t_n \Downarrow^\sharp v^\sharp, \mathscr{A}'}\ \textsc{App}$$

Figure 6: Abstract Interpretation of Skeletons and Terms

### 5.3 Abstract Interpretation of Skel

The abstract interpretation of skeletons is given on Figure 6. It maintains a callstack of specified function calls which is used to prevent infinite computations by detecting identical nested calls. A callstack is an ordered list of frames. The set of callstacks $\Pi$ is defined as:

$$\frac{}{\textbf{emp} \in \Pi}$$

$$\frac{\mathscr{A}\ \text{an AI-state} \qquad \textbf{val}\ f : \tau_1 \to .. \to \tau_n \to \tau\ =\ \mathtt{t} \in \mathscr{S} \qquad \mathrm{na}(\tau) \qquad v_i \in V^\sharp(\tau_i) \qquad \pi \in \Pi}{(f, \mathscr{A}, [v_1, .., v_n]) :: \pi \in \Pi}$$

The abstract interpretation of skeletons is similar to the big-step interpretation: the evaluation of terms is almost unchanged except that evaluating a closure or a tuple returns a singleton. When evaluating a skeleton (branch, let-binding, or application), a state of the abstract interpretation is carried through the computations.

In the BRANCH rule, all branches are evaluated and joined instead of only one branch being evaluated. Pattern matching now returns set of environments rather than a single one (explained later). As a consequence, the LETIN rule may evaluate $S_2$ in several abstract environments. This flexibility in the control-flow of the abstract interpreter allows us to do control flow analysis for $\lambda$-calculus. The APP rule evaluates all terms and pass a list of values to the application relation, defined in Figure 7.

Because the abstraction of a function is a set of closures and named closures, the APP-SET rule evaluates each one individually. The BASE rule returns the remaining value when all arguments have been

$$\frac{\begin{array}{c}\textbf{val } f : \tau_1 \to .. \to \tau_n \to \tau = t \in \mathscr{S} \\ \mathrm{na}(\tau) \qquad \emptyset, t \Downarrow v^\sharp \qquad \textbf{update}^{in}_f(\mathscr{A}, [v_1^\sharp, .., v_n^\sharp]) = \mathscr{A}_1, [v_1'^\sharp, .., v_n'^\sharp] \qquad (f, \mathscr{A}_1, [v_1'^\sharp, .., v_n'^\sharp]) \notin \pi \\ (f, [v_1'^\sharp, .., v_n'^\sharp]) :: \pi, \mathscr{A}_1, v^\sharp\ v_1'^\sharp..v_n'^\sharp \Downarrow_{app} w^\sharp, \mathscr{A}_2 \qquad \textbf{update}^{out}_f(\mathscr{A}_2, [v_1'^\sharp, .., v_n'^\sharp], w^\sharp) = \mathscr{A}_3, w'^\sharp\end{array}}{\pi, \mathscr{A}, (f,n)\ v_1^\sharp..v_n^\sharp \Downarrow_{app} w'^\sharp, \mathscr{A}_3}\ \textsc{Spec}$$

$$\frac{\begin{array}{c}\textbf{val } f : \tau_1 \to .. \to \tau_n \to \tau = t \in \mathscr{S} \qquad \mathrm{na}(\tau) \\ \emptyset, t \Downarrow v^\sharp \qquad \textbf{update}^{in}_f(\mathscr{A}, [v_1^\sharp, .., v_n^\sharp]) = \mathscr{A}_1, [v_1'^\sharp, .., v_n'^\sharp] \qquad (f, \mathscr{A}_1, [v_1'^\sharp, .., v_n'^\sharp]) \in \pi\end{array}}{\pi, \mathscr{A}, (f,n)\ v_1^\sharp..v_n^\sharp \Downarrow_{app} \bot, \mathscr{A}_1}\ \textsc{Spec-Loop}$$

$$\frac{\textbf{val } f : \tau_1 \to .. \to \tau_n \to \tau \in \mathscr{S} \qquad \mathrm{na}(\tau) \qquad [\![f]\!]^\sharp(\mathscr{A}, v_1^\sharp, .., v_n^\sharp) = w^\sharp, \mathscr{A}'}{\pi, \mathscr{A}, (f,n)\ v_1^\sharp..v_n^\sharp \Downarrow_{app} w^\sharp, \mathscr{A}'}\ \textsc{Unspec}$$

Figure 7: Abstract Interpretation: Application

processed. The CLOS rule evaluates the body of the function $S$ in all abstract environments returned by the matching of the pattern against the argument. The SPEC rule evaluates the call to a specified function and maintains invariants. Invariants depend on the analysis and the AI-state, therefore, language-dependent update functions can be provided to to maintain invariants before and after a call. The update functions must respect the following monotonicity constraints in order to ensure soundness:

**Definition 4** *The update functions are said to be monotonic if and only if:*

$$\textbf{\textit{update}}^{in}_f(\mathscr{A}, [v_1^\sharp, .., v_k^\sharp]) = \mathscr{A}', [v_1'^\sharp, .., v_k'^\sharp] \implies \mathscr{A} \sqsubseteq^\sharp \mathscr{A}' \wedge (v_1^\sharp, .., v_k^\sharp) \sqsubseteq^\sharp (v_1'^\sharp, .., v_k'^\sharp)$$
$$\textbf{\textit{update}}^{out}_f(\mathscr{A}, [v_1^\sharp, .., v_k^\sharp], v^\sharp) = \mathscr{A}', v'^\sharp \implies \mathscr{A} \sqsubseteq^\sharp \mathscr{A}' \wedge v^\sharp \sqsubseteq^\sharp v'^\sharp$$

The update functions of While are defined as:

$$\textbf{update}^{in}_{\texttt{eval\_stmt}}(\mathscr{A}, [s_i^\sharp, \mathrm{pp}]) = \mathscr{A}\{(\mathrm{pp}, \texttt{In}) \mapsto s^\sharp\}, [s^\sharp, \mathrm{pp}] \qquad s^\sharp = s_i^\sharp \nabla^\sharp \mathscr{A}(\mathrm{pp}, \texttt{In})$$
$$\textbf{update}^{out}_{\texttt{eval\_stmt}}(\mathscr{A}, [s_i^\sharp, \mathrm{pp}], s_o^\sharp) = \mathscr{A}\{(\mathrm{pp}, \texttt{Out}) \mapsto s^\sharp\}, [s^\sharp, \mathrm{pp}] \qquad s^\sharp = s_o^\sharp \sqcup^\sharp \mathscr{A}(\mathrm{pp}, \texttt{Out})$$

The update functions maintain the AI-state which holds an input and an output abstract store for each program point. $\textbf{update}^{in}_{\texttt{eval\_stmt}}(\mathscr{A}, [s_i^\sharp, \mathrm{pp}])$ updates the input abstract store at program point pp for a greater abstract store, obtained by widening to ensure termination (discussed later), that contains $s_i^\sharp$, and the call to `eval_stmt` is done with this new abstract store. $\textbf{update}^{out}_{\texttt{eval\_stmt}}(\mathscr{A}, [s_i^\sharp, \mathrm{pp}], s_o^\sharp)$ makes a similar change to the AI-state for the output store. The update functions for `eval_expr` (not presented here) do not change the argument, the result or the AI-state.

**Lemma 6** *The update functions for While previously defined are monotonic.*

The extension of environments, or pattern matching, is presented in [7] and now returns a set $\xi$ of abstract environments as the abstraction of tuples is a finite set of tuples of abstract values. We thus return one abstract environment per tuple of abstract values in our abstract tuple.

The termination of our analysis is not formally proven. Our intuition is that an infinite derivation is necessarily caused by an infinity of calls to a specified function. Given a program point, widening the stores in the input update function for `eval_st` should ensure that the input store converges and that the SPEC-LOOP rule of the abstract interpretation ends the computation, as we have reached a local fixpoint for the program point.

## 5.4 Correctness of the Abstract Interpretation

Our methodology aims at defining mathematically correct abstract interpreters from Skeletal Semantics. In this section, we present a theorem stating that the abstract interpreter of Skel computes a correct approximation of the big-step semantics of Skel.

We state the following theorem of correctness that states that the abstract interpretation of Skel computes a sound approximation of the big-step interpretation of Skel.

**Theorem 1** *Let $\mathscr{S}$ be a Skeletal Semantics with unspecified terms Te and unspecified types Ty, and let $E$ and $E^\sharp$ be a concrete and abstract environment, respectively. Suppose*

- $\forall x \in Te$, $[\![x]\!]^\sharp$ *is a sound approximation of* $[\![x]\!]^{ppt}$.
- $\forall \tau \in Ty$, $\gamma_\tau$ *is monotonic.*
- $\textbf{\textit{update}}^{in}$ *and* $\textbf{\textit{update}}^{out}$ *are monotonic.*

*Then:*

$$\left.\begin{array}{c} E \in \gamma_\Gamma(\mathscr{A}_0, E^\sharp) \\ E, S \Downarrow v \\ \textbf{\textit{emp}}, \mathscr{A}_0, E^\sharp, S \Downarrow^\sharp v^\sharp, \mathscr{A} \end{array}\right\} \implies v \in \gamma(\mathscr{A}, v^\sharp)$$

Therefore, to prove the soundness of the analysis, it is sufficient to prove that the abstract instantiation of terms are sound approximation of the concrete ones, and that the update functions and concretisation functions are monotonic.

Let $\sigma_0 \in V^{\texttt{ppt}}(\texttt{store})$ and $\sigma_0^\sharp \in V^\sharp(\texttt{store})$ be the concrete and abstract stores with empty domain. Let $E_0 = \{ s \mapsto \sigma_0, t \mapsto \underline{\varepsilon} \}$ and $E_0^\sharp = \left\{ s \mapsto \sigma_0^\sharp, t \mapsto \underline{\varepsilon} \right\}$ be a concrete and an abstract Skel environments. We recall that $\underline{\varepsilon}$ is the program point of the root of **prg**, the analysed program. Let $\mathscr{A}_0$ be the empty mapping from program points and flow tags (`In` or `Out`) to abstract stores.

**Lemma 7** $\sigma_0 \in \gamma_{store}(\mathscr{A}_0, \sigma_0^\sharp)$

**Lemma 8** *Let* $\Gamma = \{ s \mapsto \texttt{store}, t \mapsto \texttt{stmt} \}$, $E_0 \in \gamma_\Gamma(\mathscr{A}, E_0^\sharp)$.

The abstract interpreter computes an abstract store that is a correct approximation of the concrete store returned by the big-step semantics.

**Theorem 2**

$$\left.\begin{array}{c} E_0, \texttt{eval\_stmt}\ (s,t) \Downarrow^{PP} \sigma \\ \textbf{\textit{emp}}, \mathscr{A}_0, E_0^\sharp, \texttt{eval\_stmt}\ (s,t) \Downarrow^\sharp \sigma^\sharp, \mathscr{A} \end{array}\right\} \implies \sigma \in \gamma(\mathscr{A}, \sigma^\sharp)$$

As an example, take **prg** $\equiv$ `x := 0; while (x < 3) x := x + 1`. The concrete and abstract interpretations will find that

$$E_0, \texttt{eval\_stmt}\ (s,t) \Downarrow^{PP} \{ x \mapsto 3 \}$$
$$\textbf{emp}, \mathscr{A}_0, E_0^\sharp, \texttt{eval\_stmt}\ (s,t) \Downarrow^\sharp \{ x \mapsto [0, +\infty] \}, \mathscr{A}$$

In accordance with Theorem 2, we observe that $\{ x \mapsto 3 \} \in \gamma(\mathscr{A}, \{ x \mapsto [0, +\infty] \})$
The abstract interpreter returns an imprecise result. Currently, our method fails to properly take into account the guards: the conditions of loops or conditional branchings are not used to refine the abstract values. In the previous While program, the guard of the loop is not used to get a precise abstract store in or after the loop. The skeletal semantics of the While language makes it unclear how to use the guards to modify the store, as it is syntactically the same before and after the evaluation of the condition.

The precision of the analysis also depends on the skeletal semantics of the language. An easy fix for our precision issue would be to modify the type of `isZero` and `isNotZero` functions such that they have type $(\texttt{store} \times \texttt{int}) \to \texttt{store}$. The abstract instantiations of these functions could then be used to refine the abstract stores.

# 6 Related Work

Our work is part of a large research effort to define sound analyses and build correct abstract interpreters from semantic description of languages. At its core, our approach is the Abstract Interpretation [4, 5] of a semantic meta-language. Abstract Interpretation is a method designed by Cousot and Cousot to define sound static analyses from a concrete semantics. In his Marktoberdorf lectures [3], Cousot describes a systematic way to derive an abstract interpretation of an imperative language from a concrete semantics and mathematically proved sound. We chose to define the Abstract Interpretation of Skel, as it is designed to mechanise semantics of languages. The benefit of analysing a meta-language is that a large part of the work to define and prove the correctness of the analysis is done once for every semantics mechanised with Skel. However, it is often less precise than defining a language specific abstract interpretation. Moreover, there have been several papers describing methods to derive abstract interpretation from different types of concrete semantics [4, 20, 13], we chose to derive abstract interpreters from a big-step semantics of Skel.

Schmidt [20] shows how to define an abstract interpretation for $\lambda$-calculus from a big-step semantics defined co-inductively. The abstract interpretation of Skel and its correctness proof follow the methods described in the paper. However Skel has more complex constructs than $\lambda$-calculus, especially branches. Moreover, the big-step of Skel is defined inductively, thus reasoning about non-terminating program is not possible. Also, to prove the correctness of the abstract interpretation of Skel, we relate the big-step derivation tree to the abstract derivation tree, similarly to Schmidt, but a key difference is that our proof is inductive when Schmidt's proof is co-inductive.

Lim and Reps propose the TSL system [12]: a tool to define machine-code instruction set and abstract interpretations. The specification of an instruction set in TSL is compiled into a Common Intermediate Representation (CIR). An abstract interpretation is defined on the CIR, therefore an abstract interpreter is derivable from any instruction set description. However, the TSL system is aimed at specifying and analysing machine code, and not languages in general. Moreover, it is unclear how it would be possible to define analyses on languages with more complex control-flow, like $\lambda$-calculus.

In the paper on Skeletal semantics, Bodin *et al.* [1] used skeletal semantics to relate concrete and abstract interpretations in order to prove correctness. An important difference between that work and the present is that their resulting abstract semantics is not computable, whereas our abstract interpretation can be executed as an analysis, as demonstrated by our implementation [19]. Moreover, our method computes an AI-state that collects information throughout the interpretation and allows to use widening using the update functions, rather than computing an Input/Output relation.

The idea of defining an abstract interpreter of a meta-language to define analyses for languages has been explored, for example by Keidel, Poulsen and Erdweg [9]. They use arrows [6] as meta-language to describe interpreters. The concrete and abstract interpreters share code using the unified interface of arrows. By instantiating language-dependent parts for the concrete interpretation and the abstract interpretation, they obtain two interpreters that can be proven sound compositionally by proving that the abstract language-dependent parts are sound approximation of the concrete language-dependent parts, similarly to Skel. However, we chose to use a dedicated meta-language, Skel, as its library [16] makes

defining interpreters for Skel convenient and one objective is to use the NecroCoq tool [15] to generate mechanised proofs that our derived abstract interpreters are correct.

## 7 Conclusion

In this paper, we propose a methodology for mechanically deriving correct abstract interpreters from mechanised semantics. Our approach is based on Skeletal Semantics and its meta-language Skel, used to write a semantic description of a language. It consists of two independent parts. First, we define an abstract interpreter for Skel which is target language-agnostic and is the core of all derived abstract interpreters from Skeletal Semantics. The abstract interpreter of Skel is proved correct with respect to the operational semantics of Skel. Second, for a given target language to analyse, abstractions must be defined. The abstract domains are defined by instantiating the unspecified types and providing comparisons and abstract unions of abstract values. The semantics of the language-specific parts are defined by instantiating the unspecified terms. By combining the abstract interpreter of Skel and the abstractions of the target language, we derive a working abstract interpreter specialised for the target language, obtained by meta-interpretation of Skel. We prove a theorem which states that the abstract interpreter of the target language is correct if the abstract instantiation of the unspecified terms are sound approximation of the concrete instantiation of the unspecified terms. We illustrate our method to build abstract interpreters on two examples: a value analysis for a small imperative language, and a CFA for $\lambda$-calculus (in the long version [7]).

The approach has been evaluated by an implementation of a tool [19] to generate abstract interpreters from any skeletal semantics. It was tested on While and $\lambda$-calculus and resulted in executable, sound analyses validating the feasibility of the approach.

The current abstract interpreters that we obtain have limitations to their precision. Part of this imprecision stems from the fact that we generate abstract interpreters for any language based on an abstract interpreter for the Skel meta-language skeletal semantics. An interesting feature of the approach is that some precision can be gained in a generic fashion by improving the underlying abstract interpretation of Skel. For example, our interval analysis for While does not refine the abstract values when entering a part of the program guarded by a condition. Take `If(Equal(x, 0), Skip, Assign(x, 0))`, evaluated in store where $\{x \mapsto \top\}$. Our abstract interpreter returns state $\{x \mapsto \top\}$. Indeed, the condition can be true or false thus both branches of the if construct are evaluated but each one is computed in the store $\{x \mapsto \top\}$ because the condition is not used to refine the abstract values. This issue can be addressed, e.g., by keeping a trace of the execution in order to know if we are computing a statement guarded by a condition. Dealing with this issue at the level of the meta-language analysis benefits all generated analyses.

## References

[1] Martin Bodin, Philippa Gardner, Thomas Jensen & Alan Schmitt (2019): *Skeletal semantics and their interpretations*. *Proceedings of the ACM on Programming Languages* 3(POPL), pp. 1–31, doi:10.1145/3291651.

[2] Denis Bogdanas & Grigore Roşu (2015): *K-Java: A complete semantics of Java*. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 445–456, doi:10.1145/2676726.2676982.

[3] Patrick Cousot (1998): *The Calculational Design of a Generic Abstract Interpreter*. Marktoberdorf Course Notes.

[4] Patrick Cousot & Radhia Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In Robert M. Graham, Michael A. Harrison & Ravi Sethi, editors: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, ACM, pp. 238–252, doi:10.1145/512950.512973.

[5] Patrick Cousot & Radhia Cousot (1979): *Systematic Design of Program Analysis Frameworks*. In Alfred V. Aho, Stephen N. Zilles & Barry K. Rosen, editors: *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, ACM Press, pp. 269–282, doi:10.1145/567752.567778.

[6] John Hughes (2000): *Generalising monads to arrows*. *Science of computer programming* 37(1-3), pp. 67–111, doi:10.1016/S0167-6423(99)00023-4.

[7] Thomas Jensen, Vincent Rébiscoul & Alan Schmitt (2023): *Deriving Abstract Interpreters from Skeletal Semantics (Long Version)*. Available at https://skeletons.inria.fr/cfa/express-sos-2023-long.pdf.

[8] Jacques-Henri Jourdan (2016): *Verasco: a formally verified C static analyzer*. Ph.D. thesis, Universite Paris Diderot-Paris VII.

[9] Sven Keidel, Casper Bach Poulsen & Sebastian Erdweg (2018): *Compositional soundness proofs of abstract interpreters*. *Proceedings of the ACM on Programming Languages* 2(ICFP), pp. 1–26, doi:10.1145/3235031.

[10] Adam Khayam, Louis Noizet & Alan Schmitt (2022): *A Faithful Description of ECMAScript Algorithms*. In: *Proceedings of the 24th International Symposium on Principles and Practice of Declarative Programming*, PPDP '22, Association for Computing Machinery, New York, NY, USA, pp. 8:1–8:14, doi:10.1145/3551357.3551381.

[11] Xavier Leroy (2009): *Formal verification of a realistic compiler*. *Communications of the ACM* 52(7), pp. 107–115, doi:10.1145/1538788.1538814.

[12] Junghee Lim & Thomas Reps (2013): *TSL: A system for generating abstract interpreters and its application to machine-code analysis*. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 35(1), pp. 1–59, doi:10.1145/2450136.2450139.

[13] Flemming Nielson (1982): *A denotational framework for data flow analysis*. *Acta Informatica* 18(3), pp. 265–287, doi:10.1007/BF00263194.

[14] Flemming Nielson (1989): *Two-level semantics and abstract interpretation*. *Theoretical Computer Science* 69(2), pp. 117–242, doi:10.1016/0304-3975(89)90091-1.

[15] Louis Noizet: *Necro Gallina Generator, https://gitlab.inria.fr/skeletons/necro-coq*. Available at https://gitlab.inria.fr/skeletons/necro-coq.

[16] Louis Noizet: *Necro Library, https://gitlab.inria.fr/skeletons/necro*. Available at https://gitlab.inria.fr/skeletons/necro.

[17] Louis Noizet & Alan Schmitt (2022): *Semantics in Skel and Necro*. In: *ICTCS 2022 - Italian Conference on Theoretical Computer Science*, *CEUR Workshop Proceedings* 3284, CEUR-WS.org, Rome, Italy, pp. 99–115.

[18] Grigore Roșu & Traian Florin Șerbănuță (2010): *An overview of the K semantic framework*. *The Journal of Logic and Algebraic Programming* 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012.

[19] Vincent Rébiscoul: *Abstract Interpreter Generator, https://gitlab.inria.fr/skeletons/abstract-interpreter-generator*. Available at https://gitlab.inria.fr/skeletons/abstract-interpreter-generator.

[20] David A Schmidt (1995): *Natural-semantics-based abstract interpretation (preliminary version)*. In: *International Static Analysis Symposium*, Springer, pp. 1–18, doi:10.1007/3-540-60360-3_28.

# Parallel Pushdown Automata and Commutative Context-Free Grammars in Bisimulation Semantics (Extended Abstract)

Jos C. M. Baeten
CWI
Amsterdam, The Netherlands
Jos.Baeten@cwi.nl

Bas Luttik
Eindhoven University of Technology
Eindhoven, The Netherlands
s.p.luttik@tue.nl

A classical theorem states that the set of languages given by a pushdown automaton coincides with the set of languages given by a context-free grammar. In previous work, we proved the pendant of this theorem in a setting with interaction: the set of processes given by a pushdown automaton coincides with the set of processes given by a finite guarded recursive specification over a process algebra with actions, choice, sequencing and guarded recursion, if and only if we add sequential value passing. In this paper, we look what happens if we consider parallel pushdown automata instead of pushdown automata, and a process algebra with parallelism instead of sequencing.

## 1 Introduction

This paper contributes to our ongoing project to integrate the theory of automata and formal languages on the one hand and concurrency theory on the other hand. The integration requires a more refined view on the semantics of automata, grammars and expressions. Instead of treating automata as language acceptors, and grammars and expressions as syntactic means to specify languages, we propose to view them both as defining process graphs. The great benefit of this approach is that process graphs can be considered modulo a plethora of behavioural equivalences [18]. One can still consider language equivalence and recover the classical theory of automata and formal languages. But one can also consider finer notions such as bisimilarity, which is better suited for interacting processes.

The project started with a structural characterisation of the class of finite automata of which the processes are denoted by regular expressions up to bisimilarity [5]. The investigation of the expressiveness of regular expressions in bisimulation semantics was continued in [9]. In [10], we replaced the Turing machine as an abstract model of a computer by the Reactive Turing Machine, which has interaction as an essential ingredient. Transitions have labels to give a notion of interactivity, and we consider the resulting process graphs modulo bisimilarity rather than language equivalence. Thus a Reactive Turing Machine defines an *executable* interactive process, refining the notion of computable function.

In the same way as classical automata theory defines a hierarchy of formal languages, we obtain a hierarchy of processes. In [4], we proved that the set of processes given by a pushdown automaton coincides with the set of processes given by a finite guarded recursive specification over a process algebra with actions, choice, sequencing and guarded recursion, if and only if we add sequential value passing. Pushdown automata provide an abstract model of a computer with a memory in the form of a stack. In this paper, we consider the abstract model of a computer with a memory in the form of a bag. We consider the correspondence between parallel pushdown automata and commutative context-free grammars. In the process setting, a commutative context-free grammar is a process algebra comprising actions, choice, parallelism and recursion. We start out from the process algebra BPP, extended with constants for acceptance and non-acceptance (deadlock).

G. Caltais and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2023).
EPTCS 387, 2023, pp. 114–131, doi:10.4204/EPTCS.387.9

Then we find that in one direction, every process of a finite guarded recursive specification over this process algebra is the process of a parallel pushdown automaton, but not the other way around: there are parallel pushdown automata with a process that is not the process of any finite guarded recursive specification. This is even the case for the one-state parallel pushdown automaton of the bag itself, there is no finite guarded BPP -specification for it. If we do want to get a recursive specification for the bag, we need to give some actions priority over others, and can find a satisfactory specification over $\mathrm{BPP}_\theta$, BPP extended with the priority operator. Indeed, we can obtain a finite guarded specification over $\mathrm{BPP}_\theta$ for every one-state parallel pushdown automaton. On the other hand, there is a parallel pushdown automaton with two states that does not have a finite guarded specification over $\mathrm{BPP}_\theta$.

If we add communication with value passing to this algebra, resulting in $\mathrm{BCP}_\theta$, we do get a complete correspondence: a process is the process of a parallel pushdown automaton if and only if it is the process of a finite guarded recursive specification. We can also get this result in a setting without the priority operator, so over BCP , but then we need that the set of values can be countable, and we have also countable summation.

To conclude, we provide a characterisation of parallel pushdown processes as a regular process communicating with a bag. In the case without priority operator, we need a form of asymmetric communication.

## 2 Preliminaries

As a common semantic framework we use the notion of a *labelled transition system*.

**Definition 1.** *A* labelled transition system *is a quadruple* $(\mathscr{S},\mathscr{A},\to,\downarrow)$*, where*

1. $\mathscr{S}$ *is a set of* states*;*
2. $\mathscr{A}$ *is a set of* actions*,* $\tau\notin\mathscr{A}$ *is the* unobservable *or* silent *action;*
3. $\to\subseteq\mathscr{S}\times\mathscr{A}\cup\{\tau\}\times\mathscr{S}$ *is an* $\mathscr{A}\cup\{\tau\}$*-labelled* transition relation*; and*
4. $\downarrow\subseteq\mathscr{S}$ *is the set of* final *or* accepting *states.*

*A* process graph *is a labelled transition system with a special designated* root state $\uparrow$*, i.e., it is a quintuple* $(\mathscr{S},\mathscr{A},\to,\uparrow,\downarrow)$ *such that* $(\mathscr{S},\mathscr{A},\to,\downarrow)$ *is a labelled transition system, and* $\uparrow\in\mathscr{S}$*. We write* $s\xrightarrow{a}s'$ *for* $(s,a,s')\in\to$ *and* $s{\downarrow}$ *for* $s\in\downarrow$*.*

For $w\in\mathscr{A}^*$ we define $s\xrightarrow{\!\!w\!\!}\!\!\!\to t$ inductively, for all states $s,t,u$: first, $s\xrightarrow{\!\!\varepsilon\!\!}\!\!\!\to s$, and then, for $a\in\mathscr{A}$, if $s\xrightarrow{a}t$ and $t\xrightarrow{\!\!w\!\!}\!\!\!\to u$, then $s\xrightarrow{\!\!aw\!\!}\!\!\!\to u$, and if $s\xrightarrow{\tau}t$ and $t\xrightarrow{\!\!w\!\!}\!\!\!\to u$, then $s\xrightarrow{\!\!w\!\!}\!\!\!\to u$.

We see that $\tau$-steps do not contribute to the string $w$. We write $s\longrightarrow t$ for there exists $a\in\mathscr{A}\cup\{\tau\}$ such that $s\xrightarrow{a}t$. Similarly, we write $s\longrightarrow\!\!\!\to t$ for "there exists $w\in\mathscr{A}^*$ such that $s\xrightarrow{\!\!w\!\!}\!\!\!\to t$" and say that $t$ is *reachable* from $s$. If $s\xrightarrow{\!\!w\!\!}\!\!\!\to t$ takes at least one step, we write $s\xrightarrow{\!\!w\!\!}\!\!\!\to^{+}t$. We write $s\not\xrightarrow{a}$ if there is no $t\in\mathscr{S}$ with $s\xrightarrow{a}t$. Finally, we write $s\xrightarrow{(a)}t$ for "$s\xrightarrow{a}t$ or $a=\tau$ and $s=t$".

By considering language equivalence classes of process graphs, we recover language equivalence as a semantics, but we can also consider other equivalence relations. Notable among these is *bisimilarity*.

**Definition 2.** *Let* $(\mathscr{S},\mathscr{A},\to,\downarrow)$ *be a labelled transition system. A symmetric binary relation R on* $\mathscr{S}$ *is a* strong bisimulation *if it satisfies the following conditions for every* $s,t\in\mathscr{S}$ *such that s R t and for all* $a\in\mathscr{A}\cup\{\tau\}$*:*

1. *if* $s\xrightarrow{a}s'$ *for some* $s'\in\mathscr{S}$*, then there is a* $t'\in\mathscr{S}$ *such that* $t\xrightarrow{a}t'$ *and* $s'\,R\,t'$*; and*

2. *if $s\downarrow$, then $t\downarrow$.*

*If there is a strong bisimulation relating $s$ and $t$ we write $s \leftrightarrows t$.*

Sometimes we can use the *strong* version of bisimilarity defined above, which does not give special treatment to $\tau$-labelled transitions. In general, when we do give special treatment to $\tau$-labeled transitions, we use some form of *branching bisimulation* [21].

**Definition 3.** *Let $(\mathscr{S}, \mathscr{A}, \rightarrow, \downarrow)$ be a labelled transition system. A symmetric binary relation $R$ on $\mathscr{S}$ is a* branching bisimulation *if it satisfies the following conditions for every $s,t \in \mathscr{S}$ such that $s\ R\ t$ and for all $a \in \mathscr{A} \cup \{\tau\}$:*

1. *if $s \xrightarrow{a} s'$ for some $s' \in \mathscr{S}$, then there are states $t', t'' \in \mathscr{S}$ such that $t \xrightarrow{\varepsilon} t'' \xrightarrow{(a)} t'$, $s\ R\ t''$ and $s'\ R\ t'$; and*
2. *if $s\downarrow$, then there is a state $t' \in \mathscr{S}$ such that $t \xrightarrow{\varepsilon} t'$ and $t'\downarrow$.*

*If there is a branching bisimulation relating $s$ and $t$, we write $s \leftrightarrows_b t$.*

In this article, we use the finest branching bisimilarity called *divergence-preserving branching bisimilarity*, which was introduced in [21] (see also [20] and [23] for an overview of recent results).

**Definition 4.** *A branching bisimulation $R$ is* divergence-preserving *if for all $s,t \in \mathscr{S}$, whenever there is a infinite sequence of states $s_0, s_1, \ldots$ such that $s = s_0$, $s_i \xrightarrow{\tau} s_{i+1}$ and $s_i\ R\ t$ for all $i \geq 0$, then there is a state $t'$ with $t \xrightarrow{\varepsilon}{}^{+} t'$ and $s_i\ R\ t'$ for some $i \geq 0$. We write $s \leftrightarrows_b^{\Delta} t$ if there is a divergence-preserving branching bisimulation relating $s$ and $t$.*

**Theorem 1.** *Strong bisimilarity, branching bisimilarity and divergence-preserving branching bisimilarity are equivalence relations on labeled transition systems.*

*Proof.* See [12] and [20]. □

A *process* is a divergence-preserving branching bisimilarity equivalence class of process graphs.

## 3 Parallel Pushdown Automata

We consider an abstract model of a computer with a memory in the form of a *bag*: the bag is an unordered multiset, an element can be removed from the bag (*get*), or an element can be added to it (*put*). Moreover, we can see when an element does not occur in the bag (a *failed* get). This is somewhat different than the definition in [24], who defined parallel pushdown automata by means of rewrite systems.

We claim our definition is a more natural one, when we compare with the definition of a pushdown automaton. In a pushdown automaton, we can pop the top element of the stack, or we can observe there is no top element (i.e., the stack is empty). In a bag, on the other hand, all elements are directly accessible. We can pop (remove) any element, or observe this element does not occur. Just observing that the bag is empty, does not lead to a satisfactory theory (see [25]).

We use notation $\mathscr{D}^{\wr\!\int}$ for the set of bags with elements from $\mathscr{D}$. We use two disjoint copies of $\mathscr{D}$, $\mathscr{D}^+ = \{(d,+) \mid d \in \mathscr{D}\}$ and $\mathscr{D}^- = \{(d,-) \mid d \in \mathscr{D}\}$. We write $+d$ instead of $(d,+)$ and $-d$ instead of $(d,-)$. We denote $\mathscr{D}^{\pm} = \mathscr{D}^+ \cup \mathscr{D}^-$.

**Definition 5** (parallel pushdown automaton)**.** *A* parallel pushdown automaton *$M$ is a sextuple $(\mathscr{S}, \mathscr{A}, \mathscr{D}, \rightarrow, \uparrow, \downarrow)$ where:*

1. *$\mathscr{S}$ is a finite set of states,*

2. $\mathscr{A}$ *is a finite input alphabet,* $\tau \notin \mathscr{A}$ *is the unobservable step,*
3. $\mathscr{D}$ *is a finite data alphabet,*
4. $\rightarrow \subseteq \mathscr{S} \times (\mathscr{A} \cup \{\tau\}) \times \mathscr{D}^{\pm} \times \mathscr{D}^{\wr\int} \times \mathscr{S}$ *is a finite set of* transitions *or* steps*,*
5. $\uparrow \in \mathscr{S}$ *is the initial state, in the initial state the bag is empty,*
6. $\downarrow \subseteq \mathscr{S}$ *is the set of final or accepting states.*

If $(s,a,+d,x,t) \in \rightarrow$ with $d \in \mathscr{D}$, we write $s \xrightarrow{a[+d/x]} t$, and this means that the machine, when it is in state $s$ and $d$ is an element of the bag, can consume input symbol $a$, replace $d$ by the bag $x$ and thereby move to state $t$. On the other hand, we write $s \xrightarrow{a[-d/x]} t$, and this means that the machine, when it is in state $s$ and the bag does not contain a $d$, can consume input symbol $a$, put $x$ in the bag and thereby move to state $t$. In steps $s \xrightarrow{\tau[+d/x]} t$ and $s \xrightarrow{\tau[-d/x]} t$, no input symbol is consumed, only the bag is modified.

Notice that we defined a parallel pushdown automaton in such a way that it can be detected whether or not an element occurs in the bag.

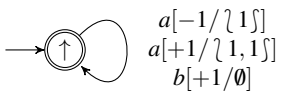



Figure 1: Parallel pushdown automaton of a counter.

For example, consider the parallel pushdown automaton depicted in Figure 1. It represents the process that can start to read an $a$, and after it has read at least one $a$, can read additional $a$'s but can also read $b$'s. Upon acceptance, it will have read up to as many $b$'s as it has read $a$'s. Interpreting symbol $a$ as an increment and $b$ as a decrement, we can see this process as a *counter*.

We do not consider the language of a parallel pushdown automaton, but rather consider the process, i.e., the divergence-preserving branching bisimilarity equivalence class of the process graph of a parallel pushdown automaton. A state of this process graph is a pair $(s,x)$, where $s \in \mathscr{S}$ is the current state and $x \in \mathscr{D}^{\wr\int}$ is the current contents of the bag. In the initial state, the bag is empty. In a final state, acceptance can take place irrespective of the contents of the bag. The transitions in the process graph are labeled by the inputs of the pushdown automaton or $\tau$.

**Definition 6.** *Let* $M = (\mathscr{S},\mathscr{A},\mathscr{D},\rightarrow,\uparrow,\downarrow)$ *be a parallel pushdown automaton. The* process graph $\mathscr{P}(M) = (\mathscr{S}_{\mathscr{P}(M)},\mathscr{A},\rightarrow_{\mathscr{P}(M)},\uparrow_{\mathscr{P}(M)},\downarrow_{\mathscr{P}(M)})$ *associated with M is defined as follows:*

1. $\mathscr{S}_{\mathscr{P}(M)} = \{(s,x) \mid s \in \mathscr{S} \;\&\; x \in \mathscr{D}^{\wr\int}\}$*;*
2. $\rightarrow_{\mathscr{P}(M)} \subseteq \mathscr{S}_{\mathscr{P}(M)} \times \mathscr{A} \cup \{\tau\} \times \mathscr{S}_{\mathscr{P}(M)}$ *is the least relation such that for all* $s,s' \in \mathscr{S}$*,* $a \in \mathscr{A} \cup \{\tau\}$*,* $d \in \mathscr{D}$ *and* $x,x' \in \mathscr{D}^{\wr\int}$ *we have*

$$(s, \wr d \int \cup x) \xrightarrow{a}_{\mathscr{P}(M)} (s', x' \cup x) \text{ if, and only if, } s \xrightarrow{a[+d/x']} s' \;;$$

$$(s,x) \xrightarrow{a}_{\mathscr{P}(M)} (s', x' \cup x) \text{ if, and only if, there exists } d \notin x \text{ such that } s \xrightarrow{a[-d/x']} s' \;;$$

3. $\uparrow_{\mathscr{P}(M)} = (\uparrow, \emptyset)$*;*

4. $\downarrow_{\mathscr{P}(M)} = \{(s,x) \mid s \in \downarrow \,\&\, x \in \mathscr{D}^{\wr\int}\}$.

To distinguish, in the definition above, the set of states, the transition relation, the initial state and the set of accepting states of the parallel pushdown automaton from similar components of the associated process graph, we have attached a subscript $\mathscr{P}(M)$ to the latter. In the remainder of this paper, we will suppress the subscript whenever it is already clear from the context whether a component of the parallel pushdown automaton or its associated process graph is meant.

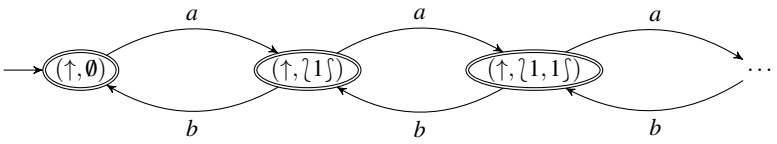



Figure 2: The process graph of the counter.

Figure 2 depicts the process graph associated with the pushdown automaton depicted in Figure 1.

In language equivalence, the definition of acceptance in parallel pushdown automata leads to the same set of languages when we define acceptance by final state (as we do here) and when we define acceptance by empty bag (not considering final states). In bisimilarity, these notions are different: acceptance by empty bag yields a smaller set of processes than acceptance by final state. Note that the process graph in Figure 2 has infinitely many non-bisimilar final states. It is, therefore, not bisimilar to the process graph of a parallel pushdown automaton that accepts by empty bag. For details, see [6, 25].

In order to illustrate that we can realise acceptance by empty bag also if we define acceptance by final state, consider the parallel pushdown automaton of the counter that only accepts when empty in Figure 3. We need three states to realise a process graph that is divergence-preserving branching bisimilar to the process graph in Figure 2, but with only the initial state accepting.

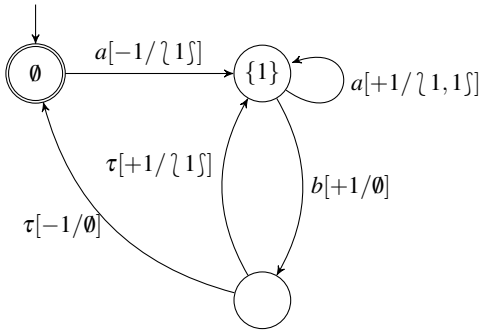



Figure 3: Counter only accepting when empty.

An important example of a parallel pushdown automaton is the bag process itself. We consider the bag that is always accepting in Figure 4. For a given data set $\mathscr{D}$, it has actions $ins(d)$ (insert), $rem(d)$ (remove) and $show(^{-}d)$ (show there is no $d$). For each $d \in \mathscr{D}$, there are the transitions shown. We need the $show(^{-}d)$ transitions later, to indicate that no (further) remove transitions are possible. We use this in Section 7.

A parallel pushdown automaton has only finitely many transitions, so there is a maximum number of transitions from a given state, called its *branching degree*. Then, also the associated process graph has a

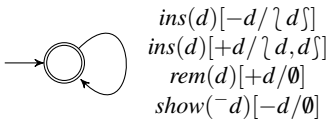



Figure 4: Parallel pushdown automaton of an always accepting bag.

branching degree, that cannot be larger than the branching degree of the underlying parallel pushdown automaton. Thus, in a process graph associated with a parallel pushdown automaton, the branching is always *bounded*. However, it is possible that its divergence-preserving branching bisimilarity equivalence class contains a process graph that is infinitely branching. Consider the following example.

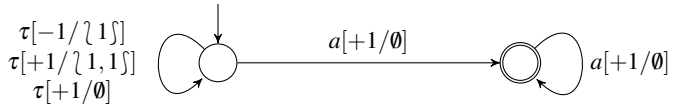



Figure 5: Parallel pushdown automaton with a divergence.

**Example 1.** *Consider the parallel pushdown automaton in Figure 5. It has a process graph consisting of two infinite rows of nodes. The nodes in the top row all have a divergence, and modulo a divergence-preserving branching bisimilarity can collaps into one node, as shown in the process graph in Figure 6. This top node still needs a divergent $\tau$ loop.*

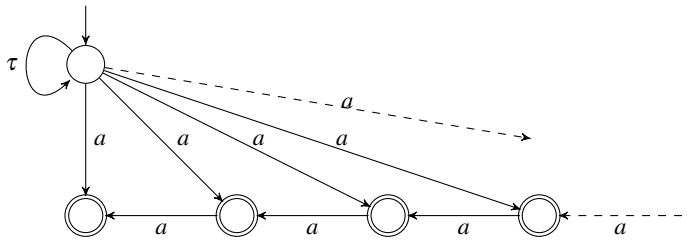



Figure 6: Process graph divergence-preserving branching bisimilar to the parallel pushdown automaton with divergence.

## 4 Parallel Processes

In the process setting, a commutative context-free grammar is a process algebra comprising actions, choice, parallelism and recursion. We start out from the process algebra PA of [14], but with sequential composition restricted to action prefixing, and then extended with constants **0** and **1** to denote deadlock and acceptance. We call this process algebra BPP$^{\mathbf{01}}$, for Basic Parallel Processes with **0** and **1**.

Let $\mathscr{A}$ be a set of *actions* and $\tau \notin \mathscr{A}$ *the silent action*, symbols denoting atomic events, and let $\mathscr{P}$ be a finite set of *process identifiers*. The sets $\mathscr{A}$ and $\mathscr{P}$ serve as parameters of the process theory that we shall introduce below. We use symbols $a, b, \ldots$, possibly indexed, to range over $\mathscr{A} \cup \{\tau\}$, symbols $X, Y, \ldots$, possibly indexed, to range over $\mathscr{P}$. The set of *parallel process expressions* is generated by the following grammar ($a \in \mathscr{A} \cup \{\tau\}$, $X \in \mathscr{P}$):

$$p ::= \mathbf{0} \mid \mathbf{1} \mid a.p \mid p + p \mid p \parallel p \mid X \quad .$$

The constants $\mathbf{0}$ and $\mathbf{1}$ respectively denote the *deadlocked* (i.e., inactive but not accepting) process and the *accepting* process. For each $a \in \mathscr{A} \cup \{\tau\}$ there is a unary action prefix operator $a.\_$. We fix a finite data set $\mathscr{D}$, and actions can be parametrised with a data element. The binary operators $+$ and $\parallel$ denote alternative composition and parallel composition, respectively. We adopt the convention that $a.\_$ binds strongest and $+$ binds weakest.

For a (possibly empty) sequence $p_1, \ldots, p_n$ we inductively define $\sum_{i=1}^{n} p_i = \mathbf{0}$ if $n = 0$ and $\sum_{i=1}^{n} p_i = (\sum_{i=1}^{n-1} p_i) + p_n$ if $n > 0$. Likewise, for a sequence $p_1, \ldots, p_n$ we inductively define $\|_{i=1}^{n} p_i = \mathbf{1}$ if $n = 0$ and $\|_{i=1}^{n} p_i = (\|_{i=1}^{n-1} p_i) \parallel p_n$ if $n > 0$.

A recursive specification over parallel process expressions is a mapping $\Gamma$ from $\mathscr{P}$ to the set of parallel process expressions. The idea is that the process expression $p$ associated with a process identifier $X \in \mathscr{P}$ by $\Gamma$ *defines* the behaviour of $X$. We prefer to think of $\Gamma$ as a collection of *defining equations* $X \stackrel{\text{def}}{=} p$, exactly one for every $X \in \mathscr{P}$. We shall, throughout the paper, presuppose a recursive specification $\Gamma$ defining the process identifiers in $\mathscr{P}$, and we shall usually simply write $X \stackrel{\text{def}}{=} p$ for $\Gamma(X) = p$. Note that, by our assumption that $\mathscr{P}$ is finite, $\Gamma$ is finite too.

$$\frac{}{\mathbf{1}\downarrow} \qquad \frac{}{a.p \xrightarrow{a} p}$$

$$\frac{p\downarrow}{(p+q)\downarrow} \qquad \frac{q\downarrow}{(p+q)\downarrow} \qquad \frac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p'} \qquad \frac{q \xrightarrow{a} q'}{p+q \xrightarrow{a} q'}$$

$$\frac{p\downarrow \quad q\downarrow}{p \parallel q\downarrow} \qquad \frac{p \xrightarrow{a} p'}{p \parallel q \xrightarrow{a} p' \parallel q} \qquad \frac{q \xrightarrow{a} q'}{p \parallel q \xrightarrow{a} p \parallel q'}$$

$$\frac{p \xrightarrow{a} p' \quad X \stackrel{\text{def}}{=} p}{X \xrightarrow{a} p'} \qquad \frac{p\downarrow \quad X \stackrel{\text{def}}{=} p}{X\downarrow}$$

Figure 7: Operational semantics for parallel process expressions.

We associate behaviour with process expressions by defining, on the set of process expressions, a unary acceptance predicate $\downarrow$ (written postfix) and, for every $a \in \mathscr{A} \cup \{\tau\}$, a binary transition relation $\xrightarrow{a}$ (written infix), by means of the transition system specification presented in Figure 7.

By means of these rules, the set of parallel process expressions turns into a labelled transition system, so we have strong bisimilarity, branching bisimilarity and divergence-preserving branching bisimilarity on parallel process expressions.

The operational rules presented in Fig 7 are in the so-called *path format* from which it immediately follows that strong bisimilarity is a congruence [11]. (Divergence-preserving) branching bisimilarity, however, is not a congruence, but by adding a rootedness condition we get rooted (divergence-preserving)

branching bisimilarity which is a congruence [21]. As we will not use equational reasoning in this paper, we will not use the rootedness condition.

Some recursive specifications over $\mathrm{BPP}^{\mathbf{01}}$ will give processes that cannot be the process of a commutative pushdown automaton.

**Example 2.** *Consider the recursive equation*

$$X \stackrel{def}{=} a.\mathbf{1} + X \parallel b.\mathbf{1} \ .$$

*We show the process graph generated by the operational rules in Figure 8. As* $X \stackrel{a}{\longrightarrow} \mathbf{1}$*, we get* $X \parallel b.\mathbf{1} \stackrel{a}{\longrightarrow} \mathbf{1} \parallel b.\mathbf{1} = b.\mathbf{1}$ *and so* $X \stackrel{a}{\longrightarrow} b.\mathbf{1}$*. Continuing like this we get* $X \stackrel{a}{\longrightarrow} b^n\mathbf{1}$ *for each n. Note we also have* $X \stackrel{b}{\longrightarrow} X$*.*

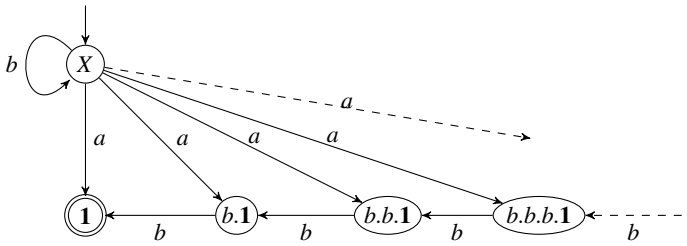



Figure 8: Process graph of the recursive specification of Example 2.

**Theorem 2.** *The process graph of Figure 8 is not divergence-preserving branching bisimilar to the process graph of any parallel pushdown automaton.*

To exclude recursive specifications over $\mathrm{BPP}^{\mathbf{01}}$ that give rise to process graphs with states that necessarily have infinitely many outgoing transitions, it suffices to formulate a standard *guardedness* condition for recursive specifications.

**Definition 7.** *We say a recursive specification is* weakly guarded *if every occurrence of a process identifier in the definition of some (possibly different) process identifier occurs within the scope of an action prefix from* $\mathscr{A} \cup \{\tau\}$*, and* strongly guarded *if every occurrence of a process identifier in the definition of some process identifier occurs within the scope of an action prefix from* $\mathscr{A}$*.*

We will show that every finite weakly guarded recursive specification over $\mathrm{BPP}^{\mathbf{01}}$ yields a parallel pushdown automaton. We first consider a couple of examples.

**Example 3.** *Consider the recursive specification*

$$AC \stackrel{def}{=} \mathbf{1} + a.(AC \parallel (\mathbf{1} + b.\mathbf{1})) \ .$$

*By following the operational rules, we obtain a process graph that is bisimilar to the one shown in Figure 2, and thus we obtain the parallel pushdown automaton in Figure 1. This is the always accepting counter.*

*If, instead, we use the equation*

$$EC \stackrel{def}{=} \mathbf{1} + a.(EC \parallel b.\mathbf{1}) \ .$$

*we get the counter that only accepts when it is empty, see the parallel pushdown automaton in Figure 3. Now we can generalize the equation of AC to the following*

$$AB \stackrel{def}{=} \mathbf{1} + \sum_{d \in \mathscr{D}} ins(d).(AB \parallel (\mathbf{1} + rem(d).\mathbf{1})) \ .$$

*We see that this is a specification of the bag. However, this bag does not have the show($^{-}d$) actions to signal that a $d$ does not occur in the bag. In fact, we will show that there is no finite weakly guarded specification over* BPP$^{\mathbf{01}}$ *that gives rise to the parallel pushdown automaton in Figure 4. For now, we first look at the other direction, to show that a finite weakly guarded specification over* BPP$^{\mathbf{01}}$ *yields the process of a parallel pushdown automaton.*

Since we have weakly guarded recursion, we can bring every BPP$^{\mathbf{01}}$-term into head normal form. The following result uses strong bisimulation, not branching bisimulation.

**Theorem 3.** *Let $\Gamma$ be a weakly guarded* BPP$^{\mathbf{01}}$*-specification. Every process expression $p$ can be brought into* head normal form*, i.e. there are $a_i \in \mathscr{A} \cup \{\tau\}$ and process expressions $p_i$ such that*

$$p \leftrightarrows (\mathbf{1}+) \sum_{i=1}^{n} a_i.p_i$$

*where the* $\mathbf{1}$ *summand may or may not occur.*

As a result, we can bring every guarded recursive specification into Greibach Normal Form.

**Definition 8.** *A recursive specification $\Gamma$ is in* Greibach Normal Form *if every equation has the form $X \stackrel{def}{=} (\mathbf{1}+) \sum_{i=1}^{n} a_i.\xi_i$ for actions $a_i \in \mathscr{A} \cup \{\tau\}$, where each $\xi_i$ is a parallel composition of identifiers of $\Gamma$, and $n \geq 0$.*

**Theorem 4.** *Let $\Gamma$ be a weakly guarded* BPP$^{\mathbf{01}}$*-specification over identifiers $\mathscr{P}$. Then there is a finite set of identifiers $\mathscr{Q}$ with $\mathscr{P} \subseteq \mathscr{Q}$ and a recursive specification in Greibach Normal Form $\Delta$ over identifiers $\mathscr{Q}$ such that for all $X, Y \in \mathscr{P}$ we have $X \leftrightarrows Y$ with respect to $\Gamma$ if, and only if, $X \leftrightarrows Y$ with respect to $\Delta$.*

Now we are ready to prove the main result of this section.

**Theorem 5.** *Every weakly guarded recursive specification over* BPP$^{\mathbf{01}}$ *has a process graph that is divergence-preserving branching bisimilar to the process graph of a parallel pushdown automaton.*

*Proof.* Without loss of generality, we can assume the specification is in Greibach Normal Form. Then, all states in the generated process graph are given by a parallel composition of identifiers of the specification. Divide the identifiers of the specification into the accepting identifiers $\mathbb{A}$ (that have a $\mathbf{1}$ summand) and the non-accepting identifiers $\mathbb{N}$ that do not have a $\mathbf{1}$ summand. A state in the generated process graph is accepting iff all identifiers in the parallel composition are from $\mathbb{A}$. In the parallel pushdown automaton to be constructed, we need to keep track when the last element of $\mathbb{N}$ is removed, in order to switch to an accepting state.

We take the data set $\mathscr{D}$ to be the set of identifiers of the specification. $S$ is the initial identifier. In the states of the parallel pushdown automaton, we will encode whether or not there is an element of $\mathbb{N}$, so there is a state for each subset (not multiset) of $\mathbb{N}$. As inspiration, we use the parallel pushdown automaton of the counter that only accepts when empty in Figure 3.

The states of the parallel pushdown automaton are as follows:

- $N$, for $N \subseteq \mathbb{N}$ (a subset, not a submultiset). The bisimulation will relate state $(N, x \cup y)$ for any multiset $x \in \mathbb{A}^{\mathfrak{V}}$ to the parallel composition of the elements of $x \cup y$, if $y$ contains all elements of $N$ and no other non-accepting identifiers.

- There is an auxiliary state $N_X$, for each $N \subseteq \mathbb{N}$ and $X \in N$.

The initial state is $\emptyset$. $\emptyset$ is the only accepting state.

Now the steps:

1. $\emptyset \xrightarrow{a[-S/\xi]} \emptyset$, whenever $S$ has a summand $a.\xi$ and $\xi$ has only accepting identifiers.
2. $\emptyset \xrightarrow{a[-S/\xi]} N$ whenever $S$ has a summand $a.\xi$ and $\xi$ has at least one non-accepting identifier. $N$ collects the non-accepting identifiers of $\xi$.
3. $N \xrightarrow{a[+X/\xi]} N'$, whenever $X \notin N$ is accepting, $X$ has a summand $a.\xi$ and $N'$ unites $N$ with the non-accepting identifiers of $\xi$.
4. $N \xrightarrow{a[+X/\xi]} N'$, whenever $X \in N$ has a summand $a.\xi$, $\xi$ has at least one non-accepting identifier and $N'$ unites $N$ with the non-accepting identifiers of $\xi$.
5. if $X \in N \subseteq \mathbb{N}$, $X$ has a summand $a.\xi$ with all identifiers in $\xi$ accepting, add three transitions

$$N \xrightarrow{a[+X/\xi]} N_X \xrightarrow{\tau[+X/\langle X \rangle]} N.$$

and

$$N_X \xrightarrow{\tau[-X/\emptyset]} N - \{X\}.$$

Notice that all the added $\tau$-steps in the transition system are inert, as from the added $N_X$ states exactly one transition can be taken, depending on whether or not $X$ occurs in the parallel composition. □

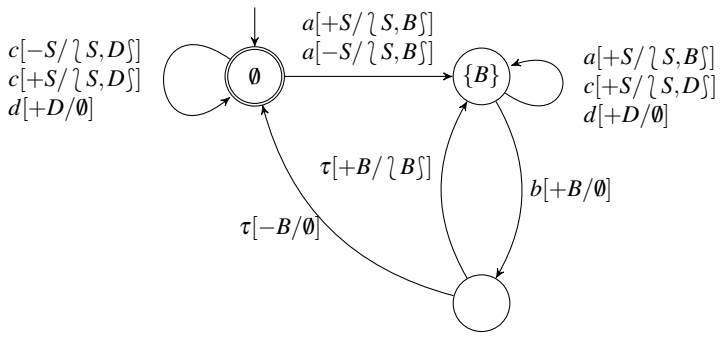



Figure 9: Parallel pushdown automaton of Example 4.

**Example 4.** *Let the guarded recursive specification* $\Gamma$ *be given as follows. Notice it is in Greibach Normal Form, and* $\mathbb{N} = \{B\}, \mathbb{A} = \{S, D\}$.

$$S \stackrel{def}{=} \mathbf{1} + a.(S \parallel B) + c.(S \parallel D) \qquad B \stackrel{def}{=} b.\mathbf{1} \qquad D \stackrel{def}{=} \mathbf{1} + d.\mathbf{1}$$

*Following the proof of Theorem 5 results in the parallel pushdown automaton shown in Figure 9. Notice the similarity with the parallel pushdown automaton shown in Figure 3.*

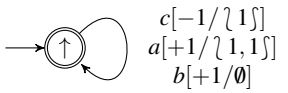



Figure 10: Parallel pushdown automaton of a counter with a change.

As we stated, the other direction does not work: we cannot find a finite weakly guarded $\mathrm{BPP}^{\mathbf{01}}$-specification for the one-state parallel pushdown automaton of the always accepting bag in Figure 4. It is technically somewhat simpler to prove such a negative result for the one-state parallel pushdown automaton shown in Figure 10.

**Theorem 6.** *For the one-state parallel pushdown automaton in Figure 10 there is no finite weakly guarded* $\mathrm{BPP}^{\mathbf{01}}$ *specification such that their process graphs are divergence-preserving branching bisimilar.*

Now let us reconsider the parallel pushdown automaton of the bag. The problem is, that a $show(^{-}d)$-action can only occur if no $rem(d)$-action can occur. Thus, in a sum context, the $show(^{-}d)$ action should have *lower priority* than the $rem(d)$ action. In general, we assume we have a partial ordering $<$ on $\mathscr{A}\cup\{\tau\}$, where $a<b$ means that $a$ has lower priority than $b$, satisfying that $\tau<a$ never holds, and whenever $a<b$ then also $a<\tau$. The *priority operator* $\theta$ will implement the priorities, and is given by the operational rules in Figure 11, see [3, 1]. Notice that the second rule for the priority operator uses a negative premise. Transition system specifications with negative premises may, in general, not define a unique transition relation that agrees with provability from the transition system specification, but our restriction to weakly guarded specifications eliminates this problem [22, 15, 19]. Also, note that (rooted) branching bisimilarity is not compatible with the priority operator, but divergence-preserving branching bisimilarity is [17].

$$\frac{p\downarrow}{\theta(p)\downarrow} \qquad \frac{p\xrightarrow{a}p' \quad \forall b>a \quad p\not\xrightarrow{b}}{\theta(p)\xrightarrow{a}\theta(p')}$$

$$\frac{p\xrightarrow{a}p'}{\rho_f(p)\xrightarrow{f(a)}\rho_f(p')} \qquad \frac{p\downarrow}{\rho_f(p)\downarrow}$$

Figure 11: Operational semantics for priorities and renaming.

With the help of this operator, we can give the following specification of the always accepting bag, assuming $show(^{-}d)<rem(d)$.

$$ABag \stackrel{\text{def}}{=} \mathbf{1}+\sum_{d\in\mathscr{D}} ins(d).\theta(ABag \parallel (\mathbf{1}+rem(d).\mathbf{1}))+\sum_{d\in\mathscr{D}} show(^{-}d).ABag \ .$$

For the parallel pushdown automaton in Figure 10, it is enough to take $c<b$. In general, we need to put a priority ordering on the labels of a parallel pushdown automaton. This may not be possible if some labels are the same, or if a $\tau$ occurs as a label. Therefore, we need to ensure all the labels in the parallel pushdown automaton are distinct and from $\mathscr{A}$, in order to be able to impose a priority ordering.

Thus, given a parallel pushdown automaton, we consider another parallel pushdown automaton with distinct labels, solve the problem for that automaton, and then rename the labels again to their original values. This renaming is done by a *renaming operator* $\rho_f$, where $f$ is any function on $\mathscr{A} \cup \{\tau\}$ satisfying $f(\tau) = \tau$. The renaming operator has the operational rules shown in Figure 11, see [2, 1].

Now we extend $\mathrm{BPP}^{\mathbf{01}}$ to include the priority operator and renaming operators. We call this extended algebra $\mathrm{BPP}^{\mathbf{01}}_{\theta}$. Theorem 5 can be extended to $\mathrm{BPP}^{\mathbf{01}}_{\theta}$.

**Theorem 7.** *Every weakly guarded recursive specification over* $\mathrm{BPP}^{\mathbf{01}}_{\theta}$ *has a process graph that is divergence-preserving branching bisimilar to the process graph of a parallel pushdown automaton.*

In the other direction, it works for every one-state parallel pushdown automaton.

**Theorem 8.** *For every one-state parallel pushdown automaton there is a finite weakly guarded* $\mathrm{BPP}^{\mathbf{01}}_{\theta}$ *specification such that their process graphs are divergence-preserving branching bisimilar.*

Thus, for all one-state parallel pushdown automata we can find a specification in $\mathrm{BPP}^{\mathbf{01}}_{\theta}$. This result does not extend to parallel pushdown automata with more than one state.

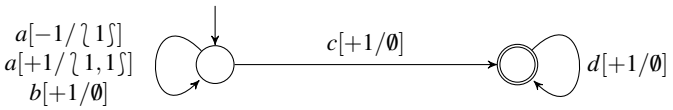



Figure 12: Parallel pushdown automaton that cannot be specified in $\mathrm{BPP}^{\mathbf{01}}_{\theta}$.

**Theorem 9.** *There is a parallel pushdown automaton with two states, such that there is no weakly guarded* $\mathrm{BPP}^{\mathbf{01}}_{\theta}$ *specification with the same process.*

In order to recover the correspondence between parallel pushdown automata and parallel process algebra, we need to add communication with value passing.

# 5 Communicating processes

We extend the basic parallel processes $\mathrm{BPP}^{\mathbf{01}}$ by adding a communication mechanism. We assume we have a finite set of communication ports $\mathscr{C}$, and that each parametrised action $c(d)$ is the result of the communication of the send action $c!d$ and the receive action $c?d$. The data set $\mathscr{D}$ is finite. Define $COM_C = \{c!d, c?d \mid c \in C, d \in \mathscr{D}\}$ for a set of ports $C \subseteq \mathscr{C}$. The *encapsulation operator* $\partial_C()$ will block the send and receive actions from the set of ports $C$. The *abstraction operator* $\tau_C$ will hide all parametrised actions from the set of ports $C$.

The process algebra $\mathrm{BCP}^{\mathbf{01}}$ extends $\mathrm{BPP}^{\mathbf{01}}$ with communication, encapsulation and abstraction; likewise, $\mathrm{BCP}^{\mathbf{01}}_{\theta}$ extends $\mathrm{BPP}^{\mathbf{01}}_{\theta}$. Using communication, we can specify communicating bags: $ABag^{io}$ defines the always accepting bag with input port $i$ and output port $o$, while $EBag^{io}$ defines the bag with input port $i$ and output port $o$ that is only accepting when it is empty. We see both the $rem(d)$ actions and the $show(^{-}d)$ actions as outputs.

$$ABag^{io} \stackrel{\mathrm{def}}{=} \mathbf{1} + \sum_{d \in \mathscr{D}} i?d.\theta(ABag^{io} \parallel (\mathbf{1} + o!(^{+}d).\mathbf{1})) + \sum_{d \in \mathscr{D}} o!(^{-}d).ABag^{io}$$

$$\frac{p \xrightarrow{c!d} p' \quad q \xrightarrow{c?d} q'}{p \parallel q \xrightarrow{c(d)} p' \parallel q' \quad q \parallel p \xrightarrow{c(d)} q' \parallel p'}$$

$$\frac{p\downarrow}{\partial_C(p)\downarrow} \qquad \frac{p \xrightarrow{a} p' \quad a \notin COM_C}{\partial_C(p) \xrightarrow{a} \partial_C(p')}$$

$$\frac{p\downarrow}{\tau_C(p)\downarrow} \qquad \frac{p \xrightarrow{c(d)} p' \quad c \in C}{\tau_C(p) \xrightarrow{\tau} \tau_C(p')} \qquad \frac{p \xrightarrow{a} p' \quad a \neq c(d) \text{ for } c \in C}{\tau_C(p) \xrightarrow{a} \tau_C(p')}$$

Figure 13: Operational semantics for communication, encapsulation and abstraction.

$$EBag^{io} \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d\in\mathscr{D}} i?d.\theta(EBag^{io} \parallel o!(^{+}d).\mathbf{1}) + \sum_{d\in\mathscr{D}} o!(^{-}d).EBag^{io} \quad .$$

In Section 7, we will use the communicating always accepting bag to make the communication between a finite control and a memory in the form of a bag explicit. Here, we restate a classical result: putting two bags with unrestricted capacity in series will again be a bag with unrestricted capacity.

$$ABag^{io} \leftrightarrows^{\Delta}_{\mathrm{b}} \tau_{\{\ell\}}(\partial_{\{\ell\}}(ABag^{i\ell} \parallel ABag^{\ell o})) \qquad EBag^{io} \leftrightarrows^{\Delta}_{\mathrm{b}} \tau_{\{\ell\}}(\partial_{\{\ell\}}(EBag^{i\ell} \parallel EBag^{\ell o}))$$

Again, we can bring every $\mathrm{BCP}^{\mathbf{01}}_{\theta}$-term into head normal form.

**Theorem 10.** *Let $\Gamma$ be a weakly guarded $\mathrm{BCP}^{\mathbf{01}}_{\theta}$-specification. Every process expression $p$ can be brought into* head normal form*, i.e. there are $a_i \in \mathscr{A} \cup \{\tau\}$ and process expressions $p_i$ such that*

$$p \leftrightarrows (\mathbf{1}+)\sum_{i=1}^{n} a_i.p_i$$

*where the $\mathbf{1}$ summand may or may not occur.*

*Proof.* By induction on the structure of $p$ (see [1]). We use Milner's Expansion Law, now with communication. □

As a result, we can bring every guarded recursive specification into Greibach Normal Form.

**Definition 9.** *A recursive specification $\Gamma$ over $\mathrm{BCP}^{\mathbf{01}}_{\theta}$ is in* Greibach Normal Form *if every equation has the form*

$$X \stackrel{def}{=} (\mathbf{1}+)\sum_{i=1}^{n} a_i.\tau_C(\partial_C(\theta(\xi_i))).$$

*for actions $a_i \in \mathscr{A} \cup \{\tau\}$, where each $\xi_i$ is a parallel composition of identifiers of $\Gamma$, and $n \geq 0$.*

**Theorem 11.** *Let $\Gamma$ be a weakly guarded $\mathrm{BCP}^{\mathbf{01}}_{\theta}$-specification over identifiers $\mathscr{P}$. Then there is a finite set of identifiers $\mathscr{Q}$ with $\mathscr{P} \subseteq \mathscr{Q}$ and a recursive specification in Greibach Normal Form $\Delta$ over identifiers $\mathscr{Q}$ such that for all $X, Y \in \mathscr{P}$ we have $X \leftrightarrows Y$ with respect to $\Gamma$ if, and only if, $X \leftrightarrows Y$ with respect to $\Delta$.*

## 6 The full correspondence

With the help of value-passing communication, we can now establish our main result: for every parallel pushdown automaton we can find a specification in $\mathrm{BCP}_\theta^{\mathbf{01}}$. The communication actions will pass on the information of the current state of the parallel pushdown automaton. Let us look at the parallel pushdown automaton in Figure 12, that did not have a finite specification in $\mathrm{BCP}_\theta^{\mathbf{01}}$.

**Example 5.** *Consider the parallel pushdown automaton in Figure 12, with initial state $s$ and accepting state $t$. We need to distinguish between the two $a$-actions on state $s$, let us call them $a^-$ and $a^+$. Action $a^-$ has lower priority than $a^+$. We just need to communicate in which of the states we are, so we use actions $p!s, p!t, p?s$ and $p?t$ for some communication port $p$. Actions $p(s), p(t)$ have the highest priority. As all components in a parallel composition need the state information, we need to communicate the state information repeatedly, until all components are brought into the right position. After this, we need to exit the communication process. Define $P_s \overset{def}{=} \mathbf{1} + p!s.P_s + exit.\mathbf{1}$ and $P_t \overset{def}{=} \mathbf{1} + p!t.P_t + exit.\mathbf{1}$ and $p(s) > exit, p(t) > exit$ and $exit > e$ for $e \in \{a^+, a^-, b, c, d\}$.*

$$S \overset{def}{=} a^-.\tau_p(\partial_p(\theta(P_s \parallel X_0 \parallel X_1)))$$

$$X_1 \overset{def}{=} p?s.(a^+.(P_s \parallel X_1 \parallel X_1) + b.P_s + c.P_t) + p?t.(\mathbf{1} + d.P_t)$$

$$X_0 \overset{def}{=} p?s.a^-.(P_s \parallel X_1) + p?t.\mathbf{1}$$

**Theorem 12.** *For every parallel pushdown automaton there is a finite weakly guarded specification over $\mathrm{BCP}_\theta^{\mathbf{01}}$ such that their process graphs are divergence-preserving branching bisimilar.*

**Theorem 13.** *For every finite weakly guarded $\mathrm{BCP}_\theta^{\mathbf{01}}$-specification there is a parallel pushdown automaton such that their process graphs are divergence-preserving branching bisimilar.*

*Proof.* As again, we can bring a finite weakly guarded $\mathrm{BCP}_\theta^{\mathbf{01}}$-specification into Greibach Normal Form, this proof goes along the lines of the proof of Theorem 5. The only difference is, is that because of a communication action, *two* non-accepting identifiers can be removed from a parallel composition at the same time. □

To conclude this section, we consider how far we can go with communication, but without priorities. In the bag, not using priorities, it is required to count the number of remove transitions, in order to know when a show absence transition is enabled. We can do this counting in the communication actions, but then the parametrising data set $\mathscr{D}$ becomes infinite, and the specification uses countable sums. This is a drawback, in our opinion.

**Theorem 14.** *For every parallel pushdown automaton there is a finite weakly guarded specification over $\mathrm{BCP}^{\mathbf{01}}$ extended with infinite choice, such that their process graphs are divergence-preserving branching bisimilar.*

**Example 6.** *Consider the parallel pushdown automaton in Figure 10, with state $s$. As the data set is a singleton, we just need to count the number of $1$'s, and we use natural numbers as parameters.*

$$S \overset{def}{=} \mathbf{1} + c.\tau_s(\partial_s(s!1.\mathbf{1} \parallel X_{\{1\}}))$$

$$X_{\{1\}} \overset{def}{=} s?1.(\mathbf{1} + (a.s!2.\mathbf{1} \parallel X_{\{1\}} \parallel X_{\{1\}}) + (b.s!0.\mathbf{1} \parallel X_\emptyset)) + \\ + \sum_{n \geq 2} s?n.(\mathbf{1} + (a.s!(n+1).\mathbf{1} \parallel X_{\{1\}} \parallel X_{\{1\}}) + b.s!(n-1).\mathbf{1})$$

$$X_\emptyset \overset{def}{=} s?0.(\mathbf{1} + (c.s!1.\mathbf{1} \parallel X_{\{1\}})).$$

# 7 A characterisation

A computer shows interaction between a finite control and the memory. The finite control can be represented by a regular process (a finite automaton). In [7], we considered a memory in the form of a stack, and we established that a pushdown process can be characterised as a regular process communicating with a stack. Here, we have a memory in the form of a bag, and we can establish a similar result.

**Theorem 15.** *A process $p$ is a parallel push-down process, if and only there is a regular process $q$ such that $p \leftrightarrows_b^{\Delta} \tau_{\{i,o\}}(\partial_{\{i,o\}}(q \parallel ABag^{io}))$.*

In [8], it was established that every parallel process expression is rooted branching bisimilar to a regular process communicating with a process that is defined as follows:

$$AB^{io} \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathscr{D}} i?d.AB^{io} \parallel (\mathbf{1} + o!d.\mathbf{1}).$$

By Theorem 5, every parallel process expression denotes a parallel pushdown process, and so Theorem 15 can be applied to get a characterisation in terms of a regular process that communicates with $ABag^{io}$, the aways accepting bag. Note, however, that $ABag^{io}$ uses the priority operator to facilitate the show absence actions. With the process $AB^{io}$ from [8] we can establish a similar result, but we have to replace receiving an element from a bag by *getting* an element from a bag, where failure to get a particular element can be detected (see [13, 1]. Thus, for $d \in \mathscr{D}$, we add elements $c??^{+}d$ (a get) and $c??^{-}d$ (a failed get) to $COM_C$ with $c \in C$, and add $c \times d$ for a failed communication that will also be hidden by $\tau_C$ with $c \in C$. We add the operational rules in Figure 14. Notice the second rule uses a negative premise. Still, as we use weakly guarded recursion, and the rules are in so-called *panth* format, we obtain a labelled transition system, see [26].

$$\frac{p \stackrel{c??^{+}d}{\longrightarrow} p' \quad q \stackrel{c!d}{\longrightarrow} q'}{p \parallel q \stackrel{c(d)}{\longrightarrow} p' \parallel q' \quad q \parallel p \stackrel{c(d)}{\longrightarrow} q' \parallel p'} \qquad \frac{p \stackrel{c??^{-}d}{\longrightarrow} p' \quad q \stackrel{c!d}{\not\rightarrow}}{p \parallel q \stackrel{c\times d}{\longrightarrow} p' \parallel q \quad q \parallel p \stackrel{c\times d}{\longrightarrow} q \parallel p'}$$

Figure 14: Operational semantics for get communication.

**Theorem 16.** *A process $p$ is a parallel push-down process, if and only there is a regular process $q$ such that $p \leftrightarrows_b^{\Delta} \tau_{\{i,o\}}(\partial_{\{i,o\}}(q \parallel AB^{io}))$.*

We see the bag is the prototypical parallel pushdown process, as all parallel pushdown processes can be realised as a regular process communicating with a bag. A bag is not a pushdown process. Likewise, the stack is the prototypical pushdown process, but not a parallel pushdown process. The counter is not a regular process, but it is both a pushdown process and a parallel pushdown process. Figure 15 provides a complete picture. The queue is not a pushdown process and also not a parallel pushdown process. It is the prototypical executable process, as every executable process can be characterized as a regular process communicating with an always accepting queue. By using a queue, the Turing tape can be defined.

# 8 Conclusion

In language theory, the set of languages given by a parallel pushdown automaton coincides with the set of languages given by a commutative context-free grammar. A language is an equivalence class

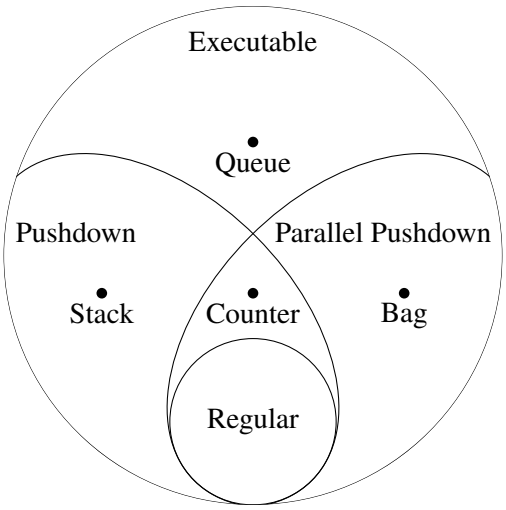



Figure 15: Classification of Executable, Pushdown, Parallel Pushdown, and Regular processes and the prototypical processes Queue, Bag, Stack and Counter.

of process graphs modulo language equivalence. A process is an equivalence class of process graphs modulo divergence-preserving branching bisimulation.

This paper solves the question how we can characterize the set of processes given by a parallel pushdown automaton. In the process setting, a commutative context-free grammar is a process algebra with actions, choice, parallel composition and finite recursion. We need to limit to weakly guarded recursion in the process setting. Starting out from the seminal process algebra PA of [14] with sequential composition restricted to action prefixing, we need to add constants for the inactive and accepting process and for the inactive non-accepting (deadlock) process. Thus, we arrive at the process algebra $\mathrm{BPP}^{\mathbf{01}}$ of the basic parallel processes. We extend this algebra with the priority operator $\theta$, in order to give some actions priority over others.

Then, every finite weakly guarded $\mathrm{BPP}^{\mathbf{01}}_{\theta}$ specification yields the process of a parallel pushdown automaton, but not the other way around, there are processes of parallel pushdown automata that cannot be given by a finite weakly guarded $\mathrm{BPP}^{\mathbf{01}}_{\theta}$ specification. For parallel pushdown automata with just one state, such a specification can be found.

We obtain a complete correspondence by adding value passing communication.

| The set of processes given by a parallel pushdown automaton coincides with the set of processes given by a finite weakly guarded recursive specification over a process algebra with actions, choice, priorities, and parallel composition with value passing communication. |
|---|

We also provide another characterisation of parallel pushdown processes: a process is a parallel pushdown process if and only if there is a regular process such that the process is divergence-preserving branching bisimilar to the regular process communicating with an always accepting bag.

This paper contributes to our ongoing project to integrate automata theory and process theory. As a result, we can present the foundations of computer science using a computer model with interaction. Such a computer model relates more closely to the computers we see all around us.

As future work, we need to compare the algebra used here with Petri nets, see e.g. [16].

## References


[1] J. C. M. Baeten, T. Basten & M. A. Reniers (2009): *Process Algebra: Equational Theories of Communicating Processes*. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, doi:10.1017/CBO9781139195003.

[2] Jos C. M. Baeten & Jan A. Bergstra (1988): *Global Renaming Operators in Concrete Process Algebra*. *Inf. Comput.* 78(3), pp. 205–245, doi:10.1016/0890-5401(88)90027-2.

[3] Jos C. M. Baeten, Jan A. Bergstra & Jan Willem Klop (1986): *Syntax and Defining Equations for an Interrupt Mechanism in Process Algebra*. *Fundamenta Informaticae* 9, pp. 127–168, doi:10.3233/FI-1986-9202.

[4] Jos C. M. Baeten, Cesare Carissimo & Bas Luttik (2023): *Pushdown Automata and Context-Free Grammars in Bisimulation Semantics*. *Logical Methods in Computer Science* 19, pp. 15:1–15.32, doi:10.46298/LMCS-19(1:15)2023.

[5] Jos C. M. Baeten, Flavio Corradini & Clemens Grabmayer (2007): *A characterization of regular expressions under bisimulation*. *J. ACM* 54(2), p. 6, doi:10.1145/1219092.1219094.

[6] Jos C. M. Baeten, Pieter Cuijpers, Bas Luttik & Paul van Tilburg (2009): *A Process-Theoretic Look at Automata*. In Farhad Arbab & Marjan Sirjani, editors: *Fundamentals of Software Engineering, Third IPM International Conference, FSEN 2009, Kish Island, Iran, April 15-17, 2009, Revised Selected Papers*, *Lecture Notes in Computer Science* 5961, Springer, pp. 1–33, doi:10.1007/978-3-642-11623-0_1.

[7] Jos C. M. Baeten, Pieter J. L. Cuijpers & P. J. A. van Tilburg (2008): *A Context-Free Process as a Pushdown Automaton*. In Franck van Breugel & Marsha Chechik, editors: *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, *Lecture Notes in Computer Science* 5201, Springer, pp. 98–113, doi:10.1007/978-3-540-85361-9_11.

[8] Jos C. M. Baeten, Pieter J. L. Cuijpers & Paul J. A. van Tilburg (2008): *A Basic Parallel Process as a Parallel Pushdown Automaton*. In Thomas T. Hildebrandt & Daniele Gorla, editors: *Proceedings of the 15th Workshop on Expressiveness in Concurrency, EXPRESS 2008, Toronto, ON, Canada, August 23, 2008*, *Electronic Notes in Theoretical Computer Science* 242, Elsevier, pp. 35–48, doi:10.1016/j.entcs.2009.06.012.

[9] Jos C. M. Baeten, Bas Luttik, Tim Muller & Paul J. A. van Tilburg (2016): *Expressiveness modulo Bisimilarity of Regular Expressions with Parallel Composition*. *Mathematical Structures in Computer Science* 26, pp. 933–968, doi:10.1017/S0960129514000309.

[10] Jos C. M. Baeten, Bas Luttik & Paul van Tilburg (2013): *Reactive Turing machines*. *Inf. Comput.* 231, pp. 143–166, doi:10.1016/j.ic.2013.08.010.

[11] Jos C. M. Baeten & Chris Verhoef (1993): *A Congruence Theorem for Structured Operational Semantics with Predicates*. In Eike Best, editor: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, *Lecture Notes in Computer Science* 715, Springer, pp. 477–492, doi:10.1007/3-540-57208-2_33.

[12] Twan Basten (1996): *Branching Bisimilarity is an Equivalence Indeed!* *Inf. Process. Lett.* 58(3), pp. 141–147, doi:10.1016/0020-0190(96)00034-8.

[13] Jan A. Bergstra (1985): *Put and get, primitives for synchronous unreliable message passing*. *Logic group preprint series* 3, pp. 1–14.

[14] Jan A. Bergstra & Jan Willem Klop (1984): *Process Algebra for Synchronous Communication*. *Information and Control* 60(1-3), pp. 109–137, doi:10.1016/S0019-9958(84)80025-X.

[15] Roland N. Bol & Jan Friso Groote (1996): *The Meaning of Negative Premises in Transition System Specifications*. *J. ACM* 43(5), pp. 863–914, doi:10.1145/234752.234756.

[16] Jürgen Dassow, Gairatzhan Mavlankulov, Mohamed Othman, Sherzod Turaev, Mohd Selamat & R Stiebe (2012): *Grammars Controlled by Petri Nets*. In Pawel Pawlewski, editor: *Petri Nets*, chapter 15, IntechOpen, Rijeka, pp. 337–358, doi:10.5772/50637.

[17] Wan Fokkink, Rob van Glabbeek & Bas Luttik (2019): *Divide and congruence III: From decomposition of modal formulas to preservation of stability and divergence*. *Inf. Comput.* 268, doi:10.1016/j.ic.2019.104435.

[18] Rob J. van Glabbeek (1993): *The Linear Time - Branching Time Spectrum II*. In Eike Best, editor: *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, *Lecture Notes in Computer Science* 715, Springer, pp. 66–81, doi:10.1007/3-540-57208-2_6.

[19] Rob van Glabbeek (2004): *The meaning of negative premises in transition system specifications II*. *J. Log. Algebr. Program.* 60-61, pp. 229–258, doi:10.1016/j.jlap.2004.03.007.

[20] Rob van Glabbeek, Bas Luttik & Nikola Trcka (2009): *Branching Bisimilarity with Explicit Divergence*. *Fundamenta Informaticae* 93(4), pp. 371–392, doi:10.3233/FI-2009-109.

[21] Rob van Glabbeek & Peter Weijland (1996): *Branching time and abstraction in bisimulation semantics*. *Journal of the ACM* 43(3), pp. 555–600, doi:10.1145/233551.233556.

[22] Jan Friso Groote (1993): *Transition System Specifications with Negative Premises*. *Theor. Comput. Sci.* 118(2), pp. 263–299, doi:10.1016/0304-3975(93)90111-6.

[23] Bas Luttik (2020): *Divergence-Preserving Branching Bisimilarity*. In Ornela Dardha & Jurriaan Rot, editors: *Proceedings Combined 27th International Workshop on Expressiveness in Concurrency and 17th Workshop on Structural Operational Semantics, EXPRESS/SOS 2020, and 17th Workshop on Structural Operational Semantics, Online, 31 August 2020*, *EPTCS* 322, pp. 3–11, doi:10.4204/EPTCS.322.2.

[24] Faron Moller (1996): *Infinite Results*. In Ugo Montanari & Vladimiro Sassone, editors: *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings*, *Lecture Notes in Computer Science* 1119, Springer, pp. 195–216, doi:10.1007/3-540-61604-7_56.

[25] Paul J. A. van Tilburg (2011): *From computability to executability : a process-theoretic view on automata theory*. Ph.D. thesis, Mathematics and Computer Science, doi:10.6100/IR716374.

[26] Chris Verhoef (1995): *A Congruence Theorem for Structured Operational Semantics with Predicates and Negative Premises*. *Nord. J. Comput.* 2(2), pp. 274–302.

# Quantifying Masking Fault-Tolerance via Fair Stochastic Games*

Pablo F. Castro
Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Argentina
Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina
pcastro@dc.exa.unrc.edu.ar

Pedro R. D'Argenio
FAMAF, Universidad Nacional de Córdoba, Córdoba, Argentina
Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina
pedro.dargenio@unc.edu.ar

Ramiro Demasi
FAMAF, Universidad Nacional de Córdoba, Córdoba, Argentina
Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina
rdemasi@unc.edu.ar

Luciano Putruele
Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Argentina
Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina
lputruele@dc.exa.unrc.edu.ar

We introduce a formal notion of masking fault-tolerance between probabilistic transition systems using stochastic games. These games are inspired in bisimulation games, but they also take into account the possible faulty behavior of systems. When no faults are present, these games boil down to probabilistic bisimulation games. Since these games could be infinite, we propose a symbolic way of representing them so that they can be solved in polynomial time. In particular, we use this notion of masking to quantify the level of masking fault-tolerance exhibited by almost-sure failing systems, i.e., those systems that eventually fail with probability 1. The level of masking fault-tolerance of almost-sure failing systems can be calculated by solving a collection of functional equations. We produce this metric in a setting in which one of the player behaves in a strong fair way (mimicking the idea of fair environments).

## 1 Introduction

Fault-tolerance [20] is an important aspect of critical systems, in which a fault may lead to important economic, or human life, losses. Examples are ubiquitous: banking systems, automotive software, communication protocols, etc. Fault-tolerant systems typically use some kind of mechanism based on redundancy such as data replication, duplicated messages and voting. However, these techniques do not consistently enhance the ability of systems to effectively tolerate faults as one could expect. Hence, quantifying the

*This work was supported by ANPCyT PICT-2017-3894 (RAFTSys), ANPCyT PICT 2019-3134, SeCyT-UNC 33620180100354CB (ARES), and EU Horizon 2020 MSCA grant agreement 101008233 (MISSION).

G. Caltais and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2023).
EPTCS 387, 2023, pp. 132–148, doi:10.4204/EPTCS.387.10

effectiveness of fault-tolerance mechanisms is an important issue when developing critical software. Additionally, in most cases, faults have a probabilistic nature, thus any technique designed for measuring system fault-tolerance should be able to cope with stochastic phenomena.

In this paper we provide a framework aimed at quantifying the fault-tolerance exhibited by concurrent probabilistic systems. This encompasses the probability of occurrence of faults as well as the use of randomized algorithms. Particularly, we focus on the so-called *masking fault-tolerance*, in which both the safety and liveness properties are preserved by the system under the occurrence of faults [20]. Intuitively, faults are masked in such a way that their occurrence cannot be observed by the users. This is often acknowledged as the most desirable kind of fault-tolerance. The aim of this paper is to provide a framework for selecting a fault-tolerance mechanism over others as well as for balancing multiple mechanisms (e.g., to ponder on cost efficient hardware redundancies vs. time demanding software artifacts).

In the last years, significant progress has been made towards defining suitable metrics or distances for diverse types of quantitative models including real-time systems [23], probabilistic models [21, 15, 6, 18, 7, 2, 32, 3], and metrics for linear and branching systems [1, 34, 27, 10, 22]. Some authors have already pointed out that these metrics can be useful to reason about the robustness and correctness of a system, notions related to fault-tolerance. Here we follow the ideas introduced in [8] where masking fault-tolerance is captured by means of a tailored bisimulation game with quantitative objectives. We extend these ideas to a probabilistic setting and define a probabilistic version of this characterization of masking fault-tolerance which, in turn, we use to define a metric to compare the "degree" of masking fault tolerance provided by different mechanisms.

More specifically, we characterize probabilistic masking fault-tolerance via a tailored variant of probabilistic bisimulation (named *masking simulation*). Roughly speaking, masking simulation relates two probabilistic transition systems. One of them acts as a system specification (i.e., a nominal model), while the other one can be thought of as a fault-tolerant implementation that takes into account possible faulty behavior. The existence of a masking simulation implies that the implementation masks all faults. This relation admits a simple game characterization via a Boolean reachability game played on a stochastic game graph.

Since in practice masking fault tolerance cannot be achieved in full, the reliability of a fault tolerance mechanism can only be measured quantitatively. Thus, we reinterpret the same game with quantitative objectives. While previously we dealt with a Boolean reachability objective, here we introduce *milestones* indicating successful progress of the model and change the objective of the game to be the expected total collected milestones. Therefore, we transform the game into an expected total reward game. We then take the measure of the fault-tolerant mechanism to be the solution of this expected total reward game.

In order to prove our results we have addressed several technical issues. First, the games rely on the notion of couplings between probabilistic distributions and, as a consequence, the number of vertices of their game graphs is infinite. To be able to deal with these infinite games, we introduce a symbolic representation for them where couplings are captured by means of equation systems. The size of these symbolic graphs is polynomial in the size of the input systems, which enables us to solve the (Boolean) simulation game in polynomial time.

Besides, stochastic games with expected total reward objectives are required to be almost surely stopping [19] or, more generally, almost surely stopping under fairness [9]. In our terms, this means that the game needs to be *almost surely failing under fairness*. Intuitively, these games model systems that will eventually fail with probability 1. This generalizes the idea that faults with some positive probability of occurrence will eventually occur during a long enough system execution.

As our game is of infinite nature, the results in [9] cannot be applied directly. Therefore we devise a finite discretization that allows us to partly reuse [9] and show that the value of the game is determined

and that it can be computed by solving a collection of functional equations via an adapted *value iteration* algorithm [13, 14, 11, 24]. Besides, as the game can only be solved if the game is almost surely failing under fairness we also provide a polynomial solution to solve this problem. We remark that both checking almost surely stopping under fairness and solving the game are calculated through the symbolic graph.

Summarizing, we define the notion of probabilistic masking simulation and provide its game characterization which we show decidable in polynomial time (Sec. 3). In Sec. 4 we define an extension of the games by considering rewards and provide a payoff function that collects the "milestones" achieved by the implementation. We show that these games are determined provided they are almost-surely failing under fairness, and give an algorithm to calculate the value of these games. We also give a polynomial time algorithm to decide if a game is almost-surely failing under fairness.

## 2 Preliminaries

A (discrete) *probability distribution* $\mu$ over a denumerable set $S$ is a function $\mu : S \to [0,1]$ such that $\mu(S) \triangleq \sum_{s\in S} \mu(s) = 1$. Let $\mathscr{D}(S)$ denote the set of all probability distributions on $S$. $\Delta_s \in \mathscr{D}(S)$ denotes the Dirac distribution for $s$, i.e., $\Delta_s(s) = 1$ and $\Delta_s(s') = 0$ whenever $s' \neq s$. The *support* set of $\mu$ is defined by $Supp(\mu) = \{s \mid \mu(s) > 0\}$.

A *Probabilistic Transition System* (PTS) [30] is a structure $A = (S, \Sigma, \to, s_0)$ where (i) $S$ is a denumerable set of *states* containing the *initial state* $s_0 \in S$, (ii) $\Sigma$ is a set of *actions*, and (iii) $\to \subseteq S \times \Sigma \times \mathscr{D}(S)$ is the *(probabilistic) transition relation*. We assume that there is always some transition leaving from every state. Here, we only consider finite PTSs, i.e., those in which the set of states $S$, the set of actions $\Sigma$ and the transition relation $\to$ are finite.

A distribution $w \in \mathscr{D}(S \times S')$ is a *coupling* for $(\mu, \mu')$, with $\mu \in \mathscr{D}(S)$ and $\mu' \in \mathscr{D}(S')$, if $w(S, \cdot) = \mu'$ and $w(\cdot, S') = \mu$. $\mathbb{C}(\mu, \mu')$ denotes the set of all couplings for $(\mu, \mu')$. It is worth noting that this defines a (two-way transport) polytope (i.e., a particular kind of bounded polyhedron). $\mathbb{V}(\mathbb{C}(\mu, \mu'))$ denotes the set of all vertices of the corresponding polytope. This set is finite if $S$ and $S'$ are finite. For $R \subseteq S \times S'$, we say that a coupling $w$ for $(\mu, \mu')$ *respects* $R$ if $Supp(w) \subseteq R$ (i.e., $w(s, s') > 0 \Rightarrow s \; R \; s'$). We define $R^{\#} \subseteq \mathscr{D}(S) \times \mathscr{D}(S')$ by $\mu \; R^{\#} \; \mu'$ if and only if there is an $R$-respecting coupling for $(\mu, \mu')$.

A *stochastic game graph* [12] is a tuple $\mathscr{G} = (V, E, V_1, V_2, V_{\mathsf{P}}, v_0, \delta)$, where $V$ is a set of vertices with $V_1, V_2, V_{\mathsf{P}} \subseteq V$ being a partition of $V$, $v_0 \in V$ is the initial vertex, $E \subseteq V \times V$, and $\delta : V_{\mathsf{P}} \to \mathscr{D}(V)$ is a probabilistic transition function such that, for all $v \in V_{\mathsf{P}}$ and $v' \in V$: $(v, v') \in E$ iff $v' \in Supp(\delta(v))$. $V_1$ and $V_2$ are the set of vertices where Players 1 and 2 are respectively allowed to play. If $V_{\mathsf{P}} = \emptyset$, then $\mathscr{G}$ is called a 2-player game graph. Moreover, if $V_1 = \emptyset$ or $V_2 = \emptyset$, then $\mathscr{G}$ is a *Markov Decision Process* (or MDP). Finally, in case that $V_1 = \emptyset$ and $V_2 = \emptyset$, $\mathscr{G}$ is a *Markov chain* (or MC). For all states $v \in V$ we define $Post(v) = \{v' \in V \mid (v, v') \in E\}$, the set of successors of $v$. Similarly, we define $Pre(v') = \{v \in V \mid (v, v') \in E\}$ as the set of predecessors of $v'$. We assume that $Post(v) \neq \emptyset$ for every $v \in V_1 \cup V_2$.

Given a game as defined above, a *play* is an infinite sequence $\rho = \rho_0, \rho_1, \ldots$ such that $(\rho_k, \rho_{k+1}) \in E$ for every $k \in \mathbb{N}$. The set of all plays is denoted by $\Omega$, and the set of plays starting at vertex $v$ is written $\Omega_v$. A *strategy* (or policy) for Player $i \in \{1, 2\}$ is a function $\pi_i : V^* \cdot V_i \to \mathscr{D}(V)$ that assigns a probabilistic distribution to each finite sequence of states such that $Supp(\pi_i(\rho \cdot v)) \subseteq Post(v)$ for all $\rho \in V^*$ and $v \in V_i$. The set of all the strategies for Player $i$ is named $\Pi_i$. A strategy $\pi_i$ is said to be *pure* (or *deterministic*) if, for every $\rho \in V^*$ and $v \in V_i$, $\pi_i(\rho \cdot v)$ is a Dirac distribution, and it is called *memoryless* if $\pi_i(\rho \cdot v) = \pi_i(v)$, for every $\rho \in V^*$ and $v \in V_i$. Given two strategies $\pi_1 \in \Pi_1$, $\pi_2 \in \Pi_2$ and a starting state $v$, the *result* of the game is a Markov chain, denoted by $\mathscr{G}_v^{\pi_1, \pi_2}$. As any Markov chain, $\mathscr{G}_v^{\pi_1, \pi_2}$ defines a probability measure $Prob_{\mathscr{G}, v}^{\pi_1, \pi_2}$ on the Borel $\sigma$-algebra generated by the cylinders of $\Omega$. If $\mathscr{A}$ is a measurable set in such

Borel $\sigma$-algebra, $Prob^{\pi_1,\pi_2}_{\mathscr{G},v}(\mathscr{A})$ is the probability that strategies $\pi_1$ and $\pi_2$ generate a play belonging to $\mathscr{A}$ from state $v$. It would normally be convenient to use LTL notation to define events. For instance, $\Diamond V' = \{\rho = \rho_0,\rho_1,\cdots \in \Omega \mid \exists i : \rho_i \in V'\}$ defines the event in which some state in $V'$ is reached. The outcome of the game, denoted by $out_v(\pi_1,\pi_2)$ is the set of possible paths of $\mathscr{G}_v^{\pi_1,\pi_2}$ starting at vertex $v$ (i.e., the possible plays when strategies $\pi_1$ and $\pi_2$ are used). When the initial state $v$ is fixed, we write $out(\pi_1,\pi_2)$ instead of $out_v(\pi_1,\pi_2)$.

A *Boolean objective* for $\mathscr{G}$ is a set $\Phi \subseteq \Omega$. A play $\rho$ is *winning* for Player 1 at vertex $v$ if $\rho \in \Phi$, otherwise it is winning for Player 2 (i.e., we consider *zero-sum* games). A strategy $\pi_1$ is a *sure winning strategy* for Player 1 from vertex $v$ if, for every strategy $\pi_2$ for Player 2, $out_v(\pi_1,\pi_2) \subseteq \Phi$. $\pi_1$ is said to be *almost-sure winning* if for every strategy $\pi_2$ for Player 2, we have $Prob^{\pi_1,\pi_2}_{\mathscr{G},v}(\Phi) = 1$. Sure and almost-sure winning strategies for Player 2 are defined in a similar way. Reachability games are games with Boolean objectives of the style: $\Diamond V'$, for some set $V' \subseteq V$. A standard result is that, if a reachability game has a sure winning strategy, then it has a pure memoryless sure winning strategy [12].

A *quantitative objective* is a measurable function $f : \Omega \to \mathbb{R}$. Given a measurable function we define $\mathbb{E}^{\pi_1,\pi_2}_{\mathscr{G},v}[f]$ as the expectation of function $f$ under probability $Prob^{\pi_1,\pi_2}_{\mathscr{G},v}$. The goal of Player 1 is to maximize the expected value of $f$, whereas the goal of Player 2 is to minimize it. Usually, quantitative objective functions are defined via a *reward function* $r : V \to \mathbb{R}$. The value of the game for Player 1 for strategy $\pi_1$ at vertex $v$, denoted $\mathrm{val}_1(\pi_1)(v)$, is defined as: $\mathrm{val}_1(\pi_1)(v) = \inf_{\pi_2 \in \Pi_2} \mathbb{E}^{\pi_1,\pi_2}_{\mathscr{G},v}[f]$. Furthermore, the *value of the game* for Player 1 from vertex $v$ is defined as: $\sup_{\pi_1 \in \Pi_1} \mathrm{val}_1(\pi_1)(v)$. Analogously, the value of the game for a Player 2 strategy $\pi_2$ and the value of the game for Player 2 are defined as $\mathrm{val}_2(\pi_2)(v) = \sup_{\pi_1 \in \Pi_1} \mathbb{E}^{\pi_1,\pi_2}_{\mathscr{G},v}[f]$ and $\inf_{\pi_2 \in \Pi_2} \mathrm{val}_2(\pi_2)(v)$, respectively. We say that a game is determined if both values are equal, that is, $\sup_{\pi_1 \in \Pi_1} \mathrm{val}_1(\pi_1)(v) = \inf_{\pi_2 \in \Pi_2} \mathrm{val}_2(\pi_2)(v)$, for every vertex $v$.

## 3 Probabilistic Masking Simulation

We start this section by defining a probabilistic extension of the strong masking simulation introduced in [8]. Roughly speaking, this is a variation of probabilistic bisimulation that takes into account the occurrence of faults (named masking simulation), and captures masking behavior. This relation serves as a starting point for defining our masking games. We prove that in the Boolean case, our games allows us to decide masking simulation. Since these games are infinite we provide a finite symbolic characterization of them. In Section 4, we extend these games with quantitative objectives, which allows us to quantify the level of fault-tolerance offered by an implementation.

*The relation.* In simple terms, a probabilistic masking simulation is a relation between PTSs that extends probabilistic bisimulation [28, 30] in order to account for fault masking. One of the PTSs acts as the nominal model (or specification), i.e., it describes the behavior of the system when no faults are considered, and the other one represents a possible fault-tolerant implementation of the specification, in which the occurrence of faults are taken into account via a fault tolerance mechanism acting upon them.

Probabilistic masking simulation allows one to analyze whether the implementation is able to mask the faults while preserving the behavior of the specification. More specifically, for non-faulty transitions, the relation behaves as probabilistic bisimulation, which is captured by means of couplings and relations respecting these couplings. The novel part is given by the occurrence of faults: if the implementation performs a fault, the nominal model matches it by an idle step (this represents internal fault masking mechanisms).

In the following, given a set of actions $\Sigma$, and a (finite) set of *fault labels* $\mathscr{F}$, with $\mathscr{F} \cap \Sigma = \emptyset$, we define $\Sigma_{\mathscr{F}} = \Sigma \cup \mathscr{F}$. Intuitively, the elements of $\mathscr{F}$ indicate the occurrence of a fault in a faulty

```
module NOMINAL
 b :  [0..1] init 0;
 m :  [0..1] init 0; // 0 = normal,
                     // 1 = refreshing
 [w0]   (m=0)          -> (b'= 0);
 [w1]   (m=0)          -> (b'= 1);
 [r0]   (m=0) & (b=0)  -> true;
 [r1]   (m=0) & (b=1)  -> true;
 [tick] (m=0)          -> p:  (m'= 1) +
                          (1-p):  true;
 [rfsh] (m=1)          -> (m'= 0);
endmodule
```

Figure 1: Memory cell: nominal model

```
module FAULTY
 v :  [0..3] init 0;
 s :  [0..2] init 0;  // 0 = normal, 1 = faulty,
                      // 2 = refreshing
 f :  [0..1] init 0;  // fault limiting artifact
 [w0]    (s!=2)           -> (v'= 0) & (s'= 0);
 [w1]    (s!=2)           -> (v'= 3) & (s'= 0);
 [r0]    (s!=2) & (v<=1)  -> true;
 [r1]    (s!=2) & (v>=2)  -> true;
 [tick]  (s!=2)           -> p:  (s'= 2) + q:  (s'= 1)
                             + (1-p-q):  true;
 [rfsh]  (s=2)            -> (s'=0)
                             & (v'= (v<=1) ?  0 :  3);
 [fault] (s=1) & (f<1)    -> (v'= (v<3) ?  (v+1) :  2)
                             & (s'= 0) & (f'= f+1);
 [fault] (s=1) & (f<1)    -> (v'= (v>0) ?  (v-1) :  1)
                             & (s'= 0) & (f'= f+1);
endmodule
```

Figure 2: Memory cell: fault-tolerant implementation.

implementation.

**Definition 1.** *Let $A = (S, \Sigma, \to, s_0)$ and $A' = (S', \Sigma_{\mathscr{F}}, \to', s_0')$ be two PTSs representing the nominal and the implementation model, respectively. $A'$ is* (strong) probabilistic masking fault-tolerant *with respect to $A$ iff there exists a relation $\mathbf{M} \subseteq S \times S'$ such that: (a) $s_0 \,\mathbf{M}\, s_0'$, and (b) for all $s \in S, s' \in S'$ with $s \,\mathbf{M}\, s'$ and all $e \in \Sigma$ and $F \in \mathscr{F}$ the following holds:*

*(1) if $s \xrightarrow{e} \mu$, then $s' \xrightarrow{e}' \mu'$ and $\mu \,\mathbf{M}^{\#}\, \mu'$ for some $\mu'$;*

*(2) if $s' \xrightarrow{e}' \mu'$, then $s \xrightarrow{e} \mu$ and $\mu \,\mathbf{M}^{\#}\, \mu'$ for some $\mu$;*

*(3) if $s' \xrightarrow{F}' \mu'$, then $\Delta_s \,\mathbf{M}^{\#}\, \mu'$.*

*If such a relation exists we say that $A'$ is a* (strong) probabilistic masking fault-tolerant implementation *of $A$, denoted $A \preceq_m A'$.*

Note that the relation can be encoded in terms of traditional probabilistic bisimulation as follows: saturate PTSs $A$ and $A'$ by adding self-loops $s \xrightarrow{F} \Delta_s$ and $s' \xrightarrow{F}' \Delta_{s'}$, respectively, for every $s \in S$, $s' \in S'$ and $F \in \mathscr{F}$. It follows from the definitions that these two new PTSs are probabilistic bisimilar iff $A \preceq_m A'$. As a consequence, checking $A \preceq_m A'$ is decidable in polynomial time.

**Example 1.** *Consider a memory cell storing one bit of information that periodically refreshes its value. The memory supports both write and read operations, and when it refreshes, it performs a read operation and overwrites the memory with the read value. This behaviour is captured by the nominal model of Fig. 1 using PRISM notation [25]. In this model,* `ri` *and* `wi` *(for* `i` $= 0, 1$*) represent the actions of reading and writing value* `i`*. The bit stored in the memory is saved in variable* `b`*. Action* `tick` *marks that one time unit has passed and, with probability* `p`*, it enables the refresh action (*`rfsh`*). Variable* `m` *indicates whether the system is in write/read mode, or producing a refresh.*

*A potential fault in this scenario occurs when a cell unexpectedly changes its value. In practice, the occurrence of such an error has a certain probability. A typical technique to deal with this situation is* redundancy*, e.g., using three memory bits instead of one. Then, writing operations are performed simultaneously on the three bits while reading returns the value read by majority (or* voting*). Fig. 2 shows this implementation with the occurrence of the fault implicitly modeled (ignore, for the time being, the red part). Variable* `v` *counts the votes for the value 1. In addition to enabling the refresh action, a*

`tick` *may also enable the occurrence of a fault with probability* `q`*, with* $\mathtt{p}+\mathtt{q} \leq 1$*. Variable* `s` *indicates whether the system is in normal mode (*$\mathtt{s}=0$*), in a state where a fault may occur (*$\mathtt{s}=1$*), or producing a refresh (*$\mathtt{s}=2$*). The red coloured text in Fig. 2 is an artifact to limit the number of faults to* 1*. Under this condition, relation* $\mathbf{M} = \{\langle (b,m),(v,s,f)\rangle \mid 2b \leq v \leq 2b+1 \wedge (m=1 \Leftrightarrow s=2)\}$ *is a probabilistic masking simulation (b, m, v, s, and f represent the values of variables* `b`*,* `m`*,* `v`*,* `s`*, and* `f`*, respectively.) It should be evident that, when the red coloured text is removed,* `FAULTY` *is not a masking fault-tolerant implementation of* `NOMINAL`*.*

*A characterization in terms of stochastic games.* We define a stochastic masking simulation game for any given nominal model $A = (S,\Sigma,\rightarrow,s_0)$ and implementation model $A' = (S',\Sigma_{\mathscr{F}},\rightarrow',s'_0)$. The game is similar to a bisimulation game [31], and it is played by two players, named for convenience the Refuter (R) and the Verifier (V). The Verifier wants to prove that $s \in S$ and $s' \in S'$ are probabilistic masking similar, and the Refuter intends to disprove that. The game starts from the pair of states $(s,s')$ and the following steps are repeated:

1) R chooses either a transition $s \xrightarrow{a} \mu$ from the nominal model or a transition $s' \xrightarrow{a}' \mu'$ from the implementation;

2a) If $a \notin \mathscr{F}$, V chooses a transition matching action $a$ from the opposite model, i.e., a transition $s' \xrightarrow{a}' \mu'$ if R's choice was from the nominal model, or a transition $s \xrightarrow{a} \mu$ otherwise. In addition, V chooses a coupling $w$ for $(\mu,\mu')$;

2b) If $a \in \mathscr{F}$, V can only select the Dirac distribution $\Delta_s$ and the only possible coupling $w$ for $(\Delta_s,\mu')$;

3) The successor pair of states $(t,t')$ is chosen probabilistically according to $w$.

If the play continues forever, then the Verifier wins; otherwise, the Refuter wins. (Notice, in particular, that the Verifier loses if she cannot match a transition label, since choosing an arbitrary coupling is always possible.) Step 2b is the only one that seems to differ from the usual bisimulation game. This is needed because of the asymmetry produced by the transitions labeled with faults. Intuitively, if the Refuter chooses to play a fault in the implementation, then the Verifier ought to mask the fault, thus she cannot freely move in the nominal model. Summing up, the probabilistic step of a fault can only be matched by a Dirac distribution on the corresponding state of the specification.

In the following we define the stochastic masking game graph that formalizes this idea. For this, define $\Sigma^i = \{e^i \mid e \in \Sigma\}$ containing all elements of $\Sigma$ indexed with superscript $i$.

**Definition 2.** *Let* $A = (S,\Sigma,\rightarrow,s_0)$ *and* $A' = (S',\Sigma_{\mathscr{F}},\rightarrow',s'_0)$ *be two PTSs. The 2-player* stochastic masking game graph $\mathscr{G}_{A,A'} = (V^{\mathscr{G}},E^{\mathscr{G}},V^{\mathscr{G}}_{\mathsf{R}},V^{\mathscr{G}}_{\mathsf{V}},V^{\mathscr{G}}_{\mathsf{P}},v^{\mathscr{G}}_0,\delta^{\mathscr{G}})$*, is defined as follows:*

$$
\begin{aligned}
V^{\mathscr{G}} &= V^{\mathscr{G}}_{\mathsf{R}} \cup V^{\mathscr{G}}_{\mathsf{V}} \cup V^{\mathscr{G}}_{\mathsf{P}}\text{, where:}\\
V^{\mathscr{G}}_{\mathsf{R}} &= \{(s,\text{-},s',\text{-},\text{-},\text{-},\mathsf{R}) \mid s \in S \wedge s' \in S'\} \cup \{v_{err}\}\\
V^{\mathscr{G}}_{\mathsf{V}} &= \{(s,\sigma^1,s',\mu,\text{-},\text{-},\mathsf{V}) \mid s \in S \wedge s' \in S' \wedge \sigma \in \Sigma \wedge s \xrightarrow{\sigma} \mu\} \cup\\
&\qquad \{(s,\sigma^2,s',\text{-},\mu',\text{-},\mathsf{V}) \mid s \in S \wedge s' \in S' \wedge \sigma \in \Sigma_{\mathscr{F}} \wedge s' \xrightarrow{\sigma}' \mu'\}\\
V^{\mathscr{G}}_{\mathsf{P}} &= \{(s,\text{-},s',\mu,\mu',w,\mathsf{P}) \mid s \in S \wedge s' \in S' \wedge w \in \mathbb{C}(\mu,\mu') \wedge\\
&\qquad\qquad \exists \sigma \in \Sigma_{\mathscr{F}} : (s \xrightarrow{\sigma} \mu \vee (\sigma \in \mathscr{F} \wedge \mu = \Delta_s)) \wedge s' \xrightarrow{\sigma}' \mu'\}\\
v^{\mathscr{G}}_0 &= (s_0,\text{-},s'_0,\text{-},\text{-},\text{-},\mathsf{R}) \quad \textit{(the Refuter starts playing)}\\
\delta^{\mathscr{G}} &: V^{\mathscr{G}}_{\mathsf{P}} \to \mathscr{D}(V^{\mathscr{G}}_{\mathsf{R}})\text{, defined by } \delta^{\mathscr{G}}((s,\text{-},s',\mu,\mu',w,\mathsf{P}))((t,\text{-},t',\text{-},\text{-},\text{-},\mathsf{R})) = w(t,t'),
\end{aligned}
$$

*where "-" fills an unused place, and $E^{\mathscr{G}}$ is the minimal set satisfying the following rules:*

$$s \xrightarrow{\sigma} \mu \Rightarrow \langle (s,\text{-},s',\text{-},\text{-},\text{-},\mathsf{R}),(s,\sigma^1,s',\mu,\text{-},\text{-},\mathsf{V})\rangle \in E^{\mathscr{G}} \tag{$1_1$}$$

$$s' \xrightarrow{\sigma}{}' \mu' \Rightarrow \langle (s,\text{-},s',\text{-},\text{-},\text{-},\mathsf{R}),(s,\sigma^2,s',\text{-},\mu',\text{-},\mathsf{V})\rangle \in E^{\mathscr{G}} \tag{$1_2$}$$

$$s' \xrightarrow{\sigma}{}' \mu' \wedge w \in \mathbb{C}(\mu,\mu') \Rightarrow \langle (s,\sigma^1,s',\mu,\text{-},\text{-},\mathsf{V}),(s,\text{-},s',\mu,\mu',w,\mathsf{P})\rangle \in E^{\mathscr{G}} \tag{$2a_1$}$$

$$\sigma \notin \mathscr{F} \wedge s \xrightarrow{\sigma} \mu \wedge w \in \mathbb{C}(\mu,\mu') \Rightarrow \langle (s,\sigma^2,s',\text{-},\mu',\text{-},\mathsf{V}),(s,\text{-},s',\mu,\mu',w,\mathsf{P})\rangle \in E^{\mathscr{G}} \tag{$2a_2$}$$

$$F \in \mathscr{F} \wedge w \in \mathbb{C}(\Delta_s,\mu') \Rightarrow \langle (s,F^2,s',\text{-},\mu',\text{-},\mathsf{V}),(s,\text{-},s',\Delta_s,\mu',w,\mathsf{P})\rangle \in E^{\mathscr{G}} \tag{2b}$$

$$(s,\text{-},s',\mu,\mu',w,\mathsf{P}) \in V_{\mathsf{P}}^{\mathscr{G}} \wedge (t,t') \in \mathit{Supp}(w) \Rightarrow \langle (s,\text{-},s',\mu,\mu',w,\mathsf{P}),(t,\text{-},t',\text{-},\text{-},\text{-},\mathsf{R})\rangle \in E^{\mathscr{G}} \tag{3}$$

$$v \in (V_{\mathsf{V}}^{\mathscr{G}} \cup \{v_{err}\}) \wedge (\nexists v' \neq v_{err} : \langle v,v'\rangle \in E^{\mathscr{G}}) \Rightarrow \langle v,v_{err}\rangle \in E^{\mathscr{G}} \tag{err}$$

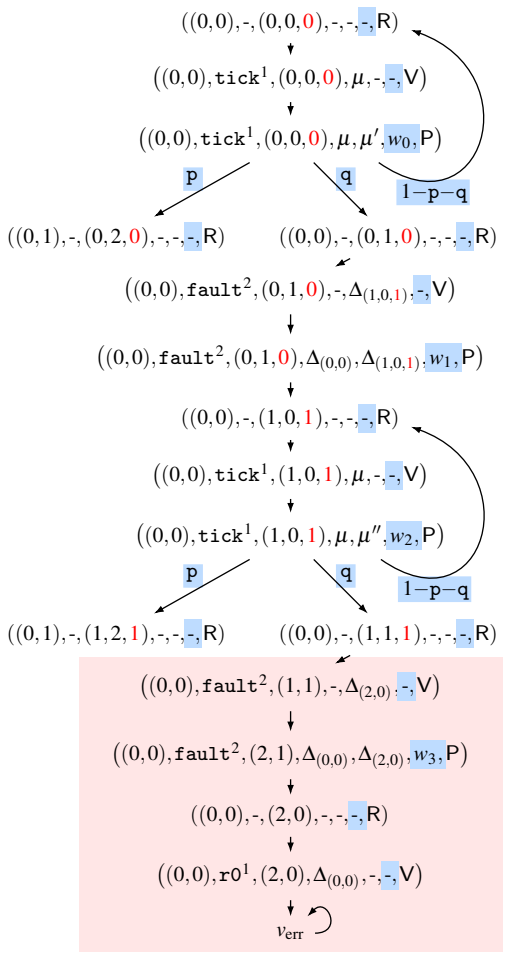
Some words about this definition are useful, it mainly follows the idea of the game previously described. A round of the game starts in the Refuter's state $v_0^{\mathscr{G}}$. Notice that, at this point, only the current states of the nominal and implementation models are relevant (all other information is not yet defined in this round and hence marked with "-"). Step 1 of the game is encoded in rules ($1_1$) and ($1_2$), where the Refuter chooses a transition, thus defining the action and distribution that need to be matched, this moves the game to a Verifier's state. A Verifier's state in $V_{\mathsf{V}}^{\mathscr{G}}$ is a tuple containing which action and distribution need to be matched, and which model the Refuter has played. Step 2a of the game is given by rules ($2a_1$) and ($2a_2$) in which the Verifier chooses a matching move from the opposite model (hence defining the other distribution) and an appropriate coupling, moving to a probabilistic state. Step 2b of the game is encoded in rule (2b). Here the Verifier has no choice since she is obliged to choose the Dirac distribution $\Delta_s$ and the only available coupling in $\mathbb{C}(\Delta_s,\mu')$. A probabilistic state in $V_{\mathsf{P}}^{\mathscr{G}}$ contains the information needed to probabilistically resolve the next step through function $\delta^{\mathscr{G}}$ (rule (3)). Finally, rule (err) states that, if a player has no move, then she reaches an error state ($v_{\text{err}}$). Note that this can only happen in a Verifier's state or in $v_{\text{err}}$.

The notion of probabilistic masking simulation can be captured by the corresponding stochastic masking game with the appropriate Boolean objective.

**Theorem 1.** *Let $A = (S,\Sigma,\to,s_0)$ and $A' = (S',\Sigma_{\mathscr{F}},\to',s_0')$ be two PTSs. Then, $A \preceq_m A'$ iff the Verifier has a sure (or almost-sure) winning strategy for the stochastic masking game graph $\mathscr{G}_{A,A'}$ with the Boolean objective $\neg\Diamond v_{err}$.*

Note that this theorem holds for both sure and almost-sure strategies of the Verifier, this follows from the fact that for stochastic reachability objectives the two kinds of strategies are equivalent.

**Example 2.** *Consider the graph in Fig. 3 (ignoring the blue shading for now). It represents a fragment of the masking game graph between* `NOMINAL` *and* `FAULTY` *of Example 1. The vertices represent the variable values in the following order:* $((\mathtt{b},\mathtt{m}),\_,(\mathtt{v},\mathtt{s},\mathtt{f}),\_,\_,\_,\_)$. *First, consider the graph disregarding the red highlighted numbers. For example,* $((0,0),\text{-},(0,0,0),\text{-},\text{-},\text{-},\mathsf{R})$ *should be read as* $((0,0),\text{-},(0,0),\text{-},\text{-},\text{-},\mathsf{R})$. *In this case we obtain the masking game graph when the red part in* `FAULTY` *is removed. Notice that, in the majority of the vertices, many outgoing edges are omitted. In particular, the Verifier vertex* $\big((0,0),\mathtt{tick}^1,(0,0),\mu,\text{-},\text{-},\mathsf{V}\big)$ *has infinitely many outgoing edges leading to probabilistic vertices of the form* $\big((0,0),\mathtt{tick}^1,(0,0),\mu,\mu',w,\mathsf{P}\big)$, *where $w$ is a coupling for $(\mu,\mu')$. In the graph, we have chosen to distinguish coupling $w_0$ which is optimal for the Verifier (similarly later for $w_2$). We highlighted the path leading to error state $v_{err}$. Notice that this occurs as a consequence of the Refuter choosing to do a second* `fault` *in vertex* $((0,0),\text{-},(1,1),\text{-},\text{-},\text{-},\mathsf{R})$ *steering the game to the red shadowed part of the graph. Later, the Refuter chooses to read 0 in the* `NOMINAL` *model (at vertex* $((0,0),\text{-},(2,0),\text{-},\text{-},\text{-},\mathsf{R})$*) which the Verifier cannot match.*




$$\mu = \mathtt{p}\cdot(0,1)+(1-\mathtt{p})\cdot(0,0)$$

$$\mu' = \begin{cases} \mathtt{p}\cdot(0,2,0)+\mathtt{q}\cdot(0,1,0)+ \\ (1-\mathtt{p}-\mathtt{q})\cdot(0,0,0) \end{cases}$$

$$\mu'' = \begin{cases} \mathtt{p}\cdot(1,2,1)+\mathtt{q}\cdot(1,1,1)+ \\ (1-\mathtt{p}-\mathtt{q})\cdot(1,0,1) \end{cases}$$

$$w_0 = \begin{cases} \mathtt{p}\cdot((0,1),(0,2,0))+\mathtt{q}\cdot((0,0),(0,1,0))+ \\ (1-\mathtt{p}-\mathtt{q})\cdot((0,0),(0,0,0)) \end{cases}$$

$$w_1 = \Delta_{((0,0),(1,0,1))}$$

$$w_2 = \begin{cases} \mathtt{p}\cdot((0,1),(1,2,1))+\mathtt{q}\cdot((0,0),(1,1,1))+ \\ (1-\mathtt{p}-\mathtt{q})\cdot((0,0),(1,0,1)) \end{cases}$$

$$w_3 = \Delta_{((0,0),(2,0))}$$

Figure 3: A fragment of a masking game graph

*Now, consider the masking game graph between* `NOMINAL` *and fault-limited* `FAULTY` *model (i.e., take now into account the red part). This graph includes the red values corresponding to variable* `f`*. Notice that here, the Refuter cannot produce a* `fault` *transition from vertex* $((0,0),\text{-},(1,1,1),\text{-},\text{-},\text{-},\mathsf{R})$*. Thus, in this case, the Verifier manages to avoid reaching the error state* $v_{err}$*.*

*A symbolic game graph.* The graph for a stochastic masking game could be infinite since each probabilistic node includes a coupling between the two contending distributions, and there can be uncountably many of them. In the following, we introduce a finite description of stochastic masking games through a symbolic representation that omits explicit reference to couplings. The definition of the symbolic game graph is twofold. The first part captures the non-stochastic behaviour of the game by removing the stochastic choice ($\delta^{\mathscr{G}}$) of the graph as well as the couplings on the vertices. The second part appends an equation system to each probabilistic vertex whose solution space is the polytope defined by the set of all couplings for the contending distributions.

**Definition 3.** *Let* $A=(S,\Sigma,\to,s_0)$ *and* $A'=(S',\Sigma_{\mathscr{F}},\to',s_0')$ *be two PTSs. The* symbolic game graph *for the stochastic masking game* $\mathscr{G}_{A,A'}$ *is defined by* $\mathscr{SG}_{A,A'}=(V^{\mathscr{SG}},E^{\mathscr{SG}},V_{\mathsf{R}}^{\mathscr{SG}},V_{\mathsf{V}}^{\mathscr{SG}},V_{\mathsf{P}}^{\mathscr{SG}},v_0^{\mathscr{SG}})$*, where:*

$V^{\mathscr{SG}}=V_{\mathsf{R}}^{\mathscr{SG}}\cup V_{\mathsf{V}}^{\mathscr{SG}}\cup V_{\mathsf{P}}^{\mathscr{SG}}$*, where:*

$$V_{\mathsf{R}}^{\mathscr{J}\mathscr{G}} = \{(s,\text{-},s',\text{-},\text{-},\mathsf{R}) \mid s \in S \wedge s' \in S'\} \cup \{v_{err}\}$$
$$V_{\mathsf{V}}^{\mathscr{J}\mathscr{G}} = \{(s,\sigma^1,s',\mu,\text{-},\mathsf{V}) \mid s \in S \wedge s' \in S' \wedge \sigma \in \Sigma \wedge s \xrightarrow{\sigma} \mu\} \cup$$
$$\{(s,\sigma^2,s',\text{-},\mu',\mathsf{V}) \mid s \in S \wedge s' \in S' \wedge \sigma \in \Sigma_{\mathscr{F}} \wedge s' \xrightarrow{\sigma}{}' \mu'\}$$
$$V_{\mathsf{P}}^{\mathscr{J}\mathscr{G}} = \{(s,\text{-},s',\mu,\mu',\mathsf{P}) \mid s \in S \wedge s' \in S' \wedge \exists \sigma \in \Sigma_{\mathscr{F}} : (s \xrightarrow{\sigma} \mu \wedge (\sigma \in \mathscr{F} \vee \mu = \Delta_s)) \wedge s' \xrightarrow{\sigma}{}' \mu'\}$$
$$v_0^{\mathscr{J}\mathscr{G}} = (s_0,\text{-},s_0',\text{-},\text{-},\mathsf{R}),$$

*and* $E^{\mathscr{J}\mathscr{G}}$ *is the minimal set satisfying the following rules:*

$$s \xrightarrow{\sigma} \mu \Rightarrow \langle (s,\text{-},s',\text{-},\text{-},\mathsf{R}),(s,\sigma^1,s',\mu,\text{-},\mathsf{V})\rangle \in E^{\mathscr{J}\mathscr{G}}$$
$$s' \xrightarrow{\sigma}{}' \mu' \Rightarrow \langle (s,\text{-},s',\text{-},\text{-},\mathsf{R}),(s,\sigma^2,s',\text{-},\mu',\mathsf{V})\rangle \in E^{\mathscr{J}\mathscr{G}}$$
$$s' \xrightarrow{\sigma}{}' \mu' \Rightarrow \langle (s,\sigma^1,s',\mu,\text{-},\mathsf{V}),(s,\text{-},s',\mu,\mu',\mathsf{P})\rangle \in E^{\mathscr{J}\mathscr{G}}$$
$$\sigma \notin \mathscr{F} \wedge s \xrightarrow{\sigma} \mu \Rightarrow \langle (s,\sigma^2,s',\text{-},\mu',\mathsf{V}),(s,\text{-},s',\mu,\mu',\mathsf{P})\rangle \in E^{\mathscr{J}\mathscr{G}}$$
$$F \in \mathscr{F} \Rightarrow \langle (s,F^2,s',\text{-},\mu',\mathsf{V}),(s,\text{-},s',\Delta_s,\mu',\mathsf{P})\rangle \in E^{\mathscr{J}\mathscr{G}}$$
$$(s,\text{-},s',\mu,\mu',\mathsf{P}) \in V_{\mathsf{P}}^{\mathscr{J}\mathscr{G}} \wedge t \in Supp(\mu) \wedge t' \in Supp(\mu') \Rightarrow \langle (s,\text{-},s',\mu,\mu',\mathsf{P}),(t,\text{-},t',\text{-},\text{-},\mathsf{R})\rangle \in E^{\mathscr{J}\mathscr{G}}$$
$$v \in (V_{\mathsf{V}}^{\mathscr{J}\mathscr{G}} \cup \{v_{err}\}) \wedge (\nexists v' \neq v_{err} : \langle v,v'\rangle \in E^{\mathscr{J}\mathscr{G}}) \Rightarrow \langle v,v_{err}\rangle \in E^{\mathscr{J}\mathscr{G}}$$

*In addition, for each* $v = (s,\text{-},s',\mu,\mu',\mathsf{P}) \in V_{\mathsf{P}}^{\mathscr{J}\mathscr{G}}$*, consider the set of variables* $X(v) = \{x_{s_i,s_j} \mid s_i \in Supp(\mu) \wedge s_j \in Supp(\mu')\}$*, and the system of equations*

$$Eq(v) = \left\{ \sum_{s_j \in Supp(\mu')} x_{s_k,s_j} = \mu(s_k) \mid s_k \in Supp(\mu) \right\} \cup$$
$$\left\{ \sum_{s_k \in Supp(\mu)} x_{s_k,s_j} = \mu'(s_j) \mid s_j \in Supp(\mu') \right\} \cup$$
$$\left\{ x_{s_k,s_j} \geq 0 \mid s_k \in Supp(\mu) \wedge s_j \in Supp(\mu') \right\}$$

Notice that $\{\bar{x}_{s_k,s_j}\}_{s_k,s_j}$ is a solution of $Eq(v)$ if and only if there is a coupling $w \in \mathbb{C}(\mu,\mu')$ such that $w(s_k,s_j) = \bar{x}_{s_k,s_j}$ for all $s_k \in Supp(\mu)$ and $s_j \in Supp(\mu')$.

Furthermore, given a set of game vertices $V' \subseteq V_{\mathsf{R}}^{\mathscr{J}\mathscr{G}}$, we define $Eq(v,V')$ by extending $Eq(v)$ with an equation limiting the couplings in such a way that vertices in $V'$ are *not* reached. Formally, $Eq(v,V') = Eq(v) \cup \left\{ \sum_{(s,\text{-},s',\text{-},\text{-},\mathsf{R}) \in V'} x_{s,s'} = 0 \right\}$. By properly defining a family of sets $V'$, we will show that the stochastic masking game can be solved in polynomial time through the symbolic game graph.

**Example 3.** *The fragment of the symbolic game graph of Example 1 in Fig. 3 is the same as depicted there only that all blue shaded components should be removed. (We also have the two variants here: one with the red values and the other one without them.) In the symbolic graph, vertex* $v = ((0,0),\texttt{tick},(0,0,0),\mu,\text{-},\mathsf{V})$*, for example, has only one successor, in contraposition to vertex* $((0,0),\texttt{tick},(0,0,0),\mu,\text{-},\text{-},\mathsf{V})$ *that has uncountably many in the original game graph. Instead,* $v$ *has associated the set* $Eq(v)$ *containing the following equations*

$$x_{(0,1),(0,2,0)} + x_{(0,1),(0,1,0)} + x_{(0,1),(0,0,0)} = \mathtt{p} \qquad x_{(0,1),(0,2,0)} + x_{(0,0),(0,2,0)} = \mathtt{p}$$
$$x_{(0,0),(0,2,0)} + x_{(0,0),(0,1,0)} + x_{(0,0),(0,0,0)} = 1-\mathtt{p} \qquad x_{(0,1),(0,0,0)} + x_{(0,0),(0,0,0)} = 1-\mathtt{p}-\mathtt{q}$$
$$x_{(0,1),(0,2,0)} \geq 0 \qquad x_{(0,1),(0,1,0)} \geq 0 \qquad x_{(0,1),(0,0,0)} \geq 0 \qquad x_{(0,1),(0,1,0)} + x_{(0,0),(0,1,0)} = \mathtt{q}$$
$$x_{(0,0),(0,2,0)} \geq 0 \qquad x_{(0,0),(0,1,0)} \geq 0 \qquad x_{(0,0),(0,0,0)} \geq 0$$

*In particular, notice that, if* $w_0$ *is as defined in Example 2,* $\bar{x}_{s,s'} = w_0(s,s')$ *is a solution for this set of equations.*

In the following we propose to use the symbolic game graph to solve the infinite game. By doing so, we obtain a polynomial time procedure. We provide an inductive construction of vertex regions $U^i$ (for $i \in \mathbb{N}$) containing the collection of vertices from which the Refuter has a strategy for reaching the error state with probability greater than 0 in at most $i$ steps.

Let $\mathscr{SG}_{A,A'} = (V^{\mathscr{SG}}, E^{\mathscr{SG}}, V_{\mathsf{R}}^{\mathscr{SG}}, V_{\mathsf{V}}^{\mathscr{SG}}, V_{\mathsf{P}}^{\mathscr{SG}}, v_0^{\mathscr{SG}})$ be a symbolic game graph for PTSs $A$ and $A'$. Define $U = \bigcup_{i\geq 0} U^i$ where, for all $i \geq 0$,

$$
\begin{aligned}
U^0 = \{v_{\text{err}}\} \qquad U^{i+1} = {} & \{v' \mid v' \in V_{\mathsf{R}}^{\mathscr{SG}} \wedge \mathit{Post}^{\mathscr{SG}}(v') \cap (\textstyle\bigcup_{j\leq i} U^j) \neq \emptyset\} \cup \\
& \{v' \mid v' \in V_{\mathsf{V}}^{\mathscr{SG}} \wedge \mathit{Post}^{\mathscr{SG}}(v') \subseteq \textstyle\bigcup_{j\leq i} U^j\} \cup \\
& \{v' \mid v' \in V_{\mathsf{P}}^{\mathscr{SG}} \wedge \mathit{Eq}(v', \mathit{Post}^{\mathscr{SG}}(v') \cap (\textstyle\bigcup_{j\leq i} U^j)) \text{ has no solution}\}
\end{aligned} \tag{1}
$$

The first line in $U^{i+1}$ corresponds to the Refuter and adds a vertex if some successor is in some previous level $U^j$. The second line corresponds to the Verifier and adds a vertex if all its successors lie in some previous $U^j$. The last line corresponds to the probabilistic player. Notice that, if $\mathit{Eq}(v', \mathit{Post}(v') \cap U^i)$ has no solution, then every possible coupling will inevitably lead with some probability to a "losing" state of a smaller level since, in particular, equation $\sum_{(s,\text{-},s',\text{-},\text{-},\mathsf{R})\in(\mathit{Post}(v')\cap U^i)} x_{s,s'} = 0$ cannot be satisfied.

The following theorem provides an algorithm to decide the stochastic masking game.

**Theorem 2.** *Let $\mathscr{G}_{A,A'}$ be a stochastic game graph for PTSs A and A′, and let $\mathscr{SG}_{A,A'}$ be the corresponding symbolic game graph. Then, the Verifier has a sure (or almost-sure) winning strategy in $\mathscr{G}_{A,A'}$ for $\neg\Diamond v_{err}$ if and only if $v_0^{\mathscr{SG}} \notin U$.*

Theorems 1 and 2 provide an alternative algorithm to decide whether there is a probabilistic masking simulation between $A$ and $A'$. This can be done in polynomial time, since $\mathit{Eq}(v,C)$ can be solved in polynomial time (e.g, using linear programming) and the number of iterations to construct $U$ is bounded by $|V^{\mathscr{SG}}|$. Since $V^{\mathscr{SG}}$ linearly depends on the transitions of the involved PTSs, the complexity is in $O(\mathit{Poly}(|\rightarrow| \cdot |\rightarrow'|))$.

# 4 Quantifying Fault Tolerance

Probabilistic masking simulation determines whether a fault-tolerant implementation is able to completely mask faults. However, in practice, this kind of masking fault-tolerance is uncommon. Usually, fault-tolerant systems are able to mask a number of faults before exhibiting a failure. In this section we extend the game theory presented above to provide a measure for the system effectiveness on masking faults. To do this, we extend the stochastic masking game with a quantitative objective function. The expected value of this function collects the (weighted) "milestones" that the fault-tolerant implementation is expected to cross before failing. A milestone is any interesting event that may occur during a system execution. For instance, a milestone may be the successful masking of a fault. In this case, the measure will reflect the number of faults that are tolerated by the system before crashing. Another milestone may be successful acknowledgments in a transmission protocol. This measures the expected number of chunks that the protocol is able to transfer before failing. Thus, milestones are some designated action labels on the implementation model and, as they may reflect different events, their value may depend on the importance of such events.

**Definition 4.** *Let $A' = (S', \Sigma_{\mathscr{F}}, \rightarrow', s'_0)$ be a PTS modeling an implementation. A* milestone *is a function* $\mathsf{m} : \Sigma_{\mathscr{F}} \rightarrow \mathbb{N}_0$.

Given a milestone $\mathsf{m}$ for $A'$, the reward $r^{\mathscr{G}}_{\mathsf{m}}$ on $\mathscr{G}_{A,A'} = (V^{\mathscr{G}}, E^{\mathscr{G}}, V^{\mathscr{G}}_{\mathsf{R}}, V^{\mathscr{G}}_{\mathsf{V}}, V^{\mathscr{G}}_{\mathsf{P}}, v^{\mathscr{G}}_0, \delta^{\mathscr{G}})$ is defined by $r^{\mathscr{G}}_{\mathsf{m}}(v) = \mathsf{m}(\sigma)$ if $v \in V^{\mathscr{G}}_{\mathsf{V}}$ and $v[1] \in \{\sigma^1, \sigma^2\}$; otherwise, $r^{\mathscr{G}}_{\mathsf{m}}(v) = 0$. Function $r^{\mathscr{G}}_{\mathsf{m}}$ collects milestones (when available) only once for each round of the game. This can be done only at Verifier's vertices since they are the only ones that save the label that it is being played in the round. The *masking payoff function* is then defined by $f_{\mathsf{m}}(\rho) = \lim_{n\to\infty}(\sum_{i=0}^{n} r^{\mathscr{G}}_{\mathsf{m}}(\rho_i))$. Therefore, the payoff function $f_{\mathsf{m}}$ represents the total of weighted milestones that a fault-tolerant implementation is able to achieve until an error state is reached. This type of payoff functions are usually called *total rewards* in the literature. One may think of this as a game played by the fault-tolerance built-in mechanism and a (malicious) player that chooses the way in which faults occur. In this game, the Verifier is the maximizer (she intends to obtain as many milestones as possible) and the Refuter is the minimizer (she intends to prevent the Verifier from collecting milestones).

Thus, the game aims to optimize $\mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v^{\mathscr{G}}_0}[f_{\mathsf{m}}]$, i.e., the expected value of random variable $f_{\mathsf{m}}$. One technical issue with total rewards objectives is that the game value may be not well-defined in $\mathbb{R}$. For instance, there could be plays not reaching an end state wherein the players collect an infinite amount of rewards. A usual condition for ensuring that the game value is well-defined is that of *almost-surely stopping*, i.e., the game has to reach a sink vertex with probability 1, for every pair of strategies [19]. In [9], we have generalized this condition to that of *almost-surely stopping under fairness*, that is, the error state $v_{\text{err}}$ is reached with probability 1 provided the Refuter plays fair. In this case the games are well-defined in $\mathbb{R}$ and determined. In simple words, determination means that the knowledge of the opponent's strategy gives no benefit for the players.

It is worth noting that fairness is necessary to prevent the Refuter from stalling the game. For instance, consider Example 1 and the stochastic masking game between the nominal and faulty models of Figs. 1 and 2 (omitting the red part). One would expect that the game leads to a failure with probability 1. However, the Refuter has strategies to avoid $v_{\text{err}}$ with positive probability. For instance, the Refuter may always play the reading action forcing the Verifier to mimic it forever and hence making the probability of reaching the error equals 0. By doing this, the Refuter stalls the game, forbidding progress and hence avoiding the occurrence of the fault. Clearly, this is against the intuitive behavior of faults which one expects will eventually occur if waiting long enough. The assumption of fairness over Refuter plays rules out this counter-intuitive behavior of the Refuter. Roughly speaking, a Refuter's fair play is one in which the Refuter commits to follow a strong fair pattern, i.e., that includes infinitely often any transition that is enabled infinitely often. Then, a fair strategy for the Refuter is a strategy that always measures 1 on the set of all the Refuter's fair plays, regardless of the strategy of the Verifier. The definitions below follow the style in [5, 4, 9].

**Definition 5.** *Given a masking game $\mathscr{G}_{A,A'} = (V^{\mathscr{G}}, E^{\mathscr{G}}, V^{\mathscr{G}}_{\mathsf{R}}, V^{\mathscr{G}}_{\mathsf{V}}, V^{\mathscr{G}}_{\mathsf{P}}, v^{\mathscr{G}}_0, \delta^{\mathscr{G}})$, the* set of all Refuter's fair plays *is defined by $RFP = \{\rho \in \Omega \mid v \in \inf(\rho) \cap V^{\mathscr{G}}_{\mathsf{R}} \Rightarrow Post(v) \subseteq \inf(\rho)\}$. A Refuter strategy $\pi_{\mathsf{R}}$ is said to be* almost-sure fair *iff, for every Verifier's strategy $\pi_{\mathsf{V}}$, $Prob^{\pi_{\mathsf{R}},\pi_{\mathsf{V}}}_{\mathscr{G},v^{\mathscr{G}}_0}(RFP) = 1$. We let $\Pi^{f}_{\mathsf{R}}$ denote the set of all fair strategies for the Refuter.*

Under this concept, the stochastic masking game is almost-sure failing under fairness if for every Verifier's strategy and every Refuter's fair strategy, the game leads to an error with probability 1. This is formally defined as follows.

**Definition 6.** *Let $A$ and $A'$ be two PTSs. We say that the stochastic masking game $\mathscr{G}_{A,A'}$ is* almost-sure failing under fairness *iff, for every strategy $\pi_{\mathsf{V}} \in \Pi_{\mathsf{V}}$ and any fair strategy $\pi_{\mathsf{R}} \in \Pi^{f}_{\mathsf{R}}$, $Prob^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v^{\mathscr{G}}_0}(\Diamond v_{err}) = 1$.*

Interestingly, under the strong fairness assumption, the determinacy of games is preserved for finite stochastic games [9]. The rest of the section is precisely devoted to bring our setting to the framework of [9] and thus provide an algorithmic solution.

A strategy $\pi_i$, $i \in \{\mathsf{R},\mathsf{V}\}$, is *semi-Markov* if for every $\hat{\rho},\hat{\rho}' \in (V^{\mathscr{G}})^*$ and $v \in V_i^{\mathscr{G}}$, $|\hat{\rho}| = |\hat{\rho}'|$ implies $\pi_i(\hat{\rho}v) = \pi_i(\hat{\rho}'v)$, that is, the decisions of $\pi_i$ depend only on the length of the run and its last state. Thus, we write $\pi_i(n,v)$ instead of $\pi_i(\hat{\rho}v)$ if $|\hat{\rho}| = n$. Let $\Pi_i^S$ denote the set of all semi-Markov strategies for Player $i$ and $\Pi_i^{Sf}$ the set of all its *fair* semi-Markov strategies.

The next lemma states that, if the Refuter plays a semi-Markov strategy, the Verifier achieves equal results regardless whether she plays an arbitrary strategy or limits to playing only semi-Markov strategies. The proof resembles that of [9, Lemma 2] taking care of the fact that the set of vertices of the stochastic masking game is uncountable. Since probabilities are anyway discrete, this is not a major technical issue, but it deserves attention in the proof.

**Lemma 1.** *Let $\mathscr{G}_{A,A'}$ be a stochastic masking game graph and let $\pi_{\mathsf{R}} \in \Pi_{\mathsf{R}}^S$ be a semi-Markov strategy. Then, for any $\pi_{\mathsf{V}} \in \Pi_{\mathsf{V}}$, there is a semi-Markov strategy $\pi_{\mathsf{V}}^* \in \Pi_{\mathsf{V}}^S$ such that $\mathbb{E}_{\mathscr{G},v}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}[f_{\mathsf{m}}] = \mathbb{E}_{\mathscr{G},v}^{\pi_{\mathsf{V}}^*,\pi_{\mathsf{R}}}[f_{\mathsf{m}}]$.*

A Verifyier strategy $\pi_{\mathsf{V}}^* \in \Pi_{\mathsf{V}}$ is *extreme* if it only moves to probabilistic vertices containing couplings that are on the polytope vertices, that is, if for all $\hat{\rho} \in (V^{\mathscr{G}})^* \times V_{\mathsf{V}}^{\mathscr{G}}$, $\pi_{\mathsf{V}}^*(\hat{\rho})((s,\text{-},s',\mu,\mu',w,\mathsf{P})) > 0$ implies that $w \in \mathbb{V}(\mathbb{C}(\mu,\mu'))$. Let $\Pi_{\mathsf{V}}^{XS}$ be the set of all extreme semi-Markov strategies for the Verifier.

Lemma 1 can be strengthened. Thus, if the Refuter plays a semi-Markov strategy, the Verifier can achieve the same result as the general case by restricting herself to play only extreme semi-Markov strategies.

**Lemma 2.** *Let $\mathscr{G}_{A,A'}$ be a stochastic masking game graph and let $\pi_{\mathsf{R}} \in \Pi_{\mathsf{R}}^S$ be a semi-Markov strategy. Then, for any $\pi_{\mathsf{V}} \in \Pi_{\mathsf{V}}^S$, there is an extreme semi-Markov strategy $\pi_{\mathsf{V}}^* \in \Pi_{\mathsf{V}}^{XS}$ such that for all $v \in V_{\mathsf{R}}^{\mathscr{G}}$, $\mathbb{E}_{\mathscr{G},v}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}[f_{\mathsf{m}}] = \mathbb{E}_{\mathscr{G},v}^{\pi_{\mathsf{V}}^*,\pi_{\mathsf{R}}}[f_{\mathsf{m}}]$.*

The key of the proof of Lemma 2 lies on the construction of $\pi_{\mathsf{V}}^*$ which is defined so that, for every $n \in \mathbb{N}$ and $v_1 \in V_{\mathsf{V}}^{\mathscr{G}}$, the probabilistic decision made by $\pi_{\mathsf{V}}^*(n,v_1)$ corresponds to a proper composition of the probabilistic decisions of $\pi_{\mathsf{V}}(n,v_1)$, and each convex combination of vertex couplings that define the coupling within each probabilistic successor $v_2 \in \mathit{Supp}(\pi_{\mathsf{V}}(n,v_1))$.

Notice that, by traveling only through probabilistic vertices on $\mathscr{G}_{A,A'}$ that are defined by vertex couplings, only a finite number of the game vertices are touched when the Verifier uses extreme strategies. Thus, we let the stochastic game graph $\mathscr{H}_{A,A'}$ be the *vertex snippet of* $\mathscr{G}_{A,A'}$ and define it to be the same as $\mathscr{G}_{A,A'}$ only that probabilistic vertices are limited to those that contain couplings in the vertices of the polytope, that is,

$$V_{\mathsf{P}}^{\mathscr{H}} = \{(s,\text{-},s',\mu,\mu',w,\mathsf{P}) \mid s \in S \wedge s' \in S' \wedge w \in \mathbb{V}(\mathbb{C}(\mu,\mu')) \wedge \\ \exists \sigma \in \Sigma_{\mathscr{F}} : (s \xrightarrow{\sigma} \mu \vee (\sigma \in \mathscr{F} \wedge \mu = \Delta_s)) \wedge s' \xrightarrow{\sigma}{}' \mu'\}.$$

The rest of the elements of $\mathscr{H}_{A,A'}$ are defined by properly restricting the domain of the respective components in $\mathscr{G}_{A,A'}$. Notice that $\mathscr{H}_{A,A'}$ is finite.

Now observe that, if the Verifier semi-Markov strategies are considered as functions with domain in $(\mathbb{N} \times V_{\mathsf{V}}^{\mathscr{G}})$, then the set of all extreme semi-Markov strategies in $\mathscr{G}_{A,A'}$ corresponds to the set of all semi-Markov strategies of $\mathscr{H}_{A,A'}$. That is: $\Pi_{\mathsf{V},\mathscr{G}}^{XS} = \Pi_{\mathsf{V},\mathscr{H}}^{S}$, where subscripts $\mathscr{G}$ and $\mathscr{H}$ indicate whether the strategies belong to $\mathscr{G}_{A,A'}$ or $\mathscr{H}_{A,A'}$, respectively. Similarly, the same holds for the set of all extreme deterministic memoryless strategies, that is: $\Pi_{\mathsf{V},\mathscr{G}}^{XMD} = \Pi_{\mathsf{V},\mathscr{H}}^{MD}$. Given the fact that $V_{\mathsf{R}}^{\mathscr{G}} = V_{\mathsf{R}}^{\mathscr{H}}$ and $V_{\mathsf{V}}^{\mathscr{G}} = V_{\mathsf{V}}^{\mathscr{H}}$, the set of all Refuter's deterministic memoryless fair strategies are the same in both game graphs, i.e., $\Pi_{\mathsf{R},\mathscr{G}}^{MDf} = \Pi_{\mathsf{R},\mathscr{H}}^{MDf}$. The following proposition follows directly from these observations.

**Proposition 1.** *Let $\mathscr{G}_{A,A'}$ be a stochastic game graph and $\mathscr{H}_{A,A'}$ its vertex snippet. Then, for all $v \in V_{\mathsf{R}}^{\mathscr{G}} (= V_{\mathsf{R}}^{\mathscr{H}})$, we have:*

1. *for all* $\pi_{\mathsf{R}} \in \Pi^{MDf}_{\mathsf{R},\mathscr{G}} (= \Pi^{MDf}_{\mathsf{R},\mathscr{H}})$, $\sup_{\pi_{\mathsf{V}} \in \Pi^{XS}_{\mathsf{V},\mathscr{G}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v}[f_{\mathsf{m}}] = \sup_{\pi_{\mathsf{V}} \in \Pi^{S}_{\mathsf{V},\mathscr{H}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{H},v}[f_{\mathsf{m}}]$; *and*
2. *for all* $\pi_{\mathsf{V}} \in \Pi^{XMD}_{\mathsf{V},\mathscr{G}} (= \Pi^{MD}_{\mathsf{V},\mathscr{H}})$, $\inf_{\pi_{\mathsf{R}} \in \Pi^{f}_{\mathsf{R},\mathscr{G}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v}[f_{\mathsf{m}}] = \inf_{\pi_{\mathsf{R}} \in \Pi^{f}_{\mathsf{R},\mathscr{H}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{H},v}[f_{\mathsf{m}}]$.

The following theorem not only states that the game for optimizing the expected value of the masking payoff function is determined, but it also guarantees that it can be solved using the finite vertex snippet of the stochastic game subgraph.

**Theorem 3.** *Let $\mathscr{G}_{A,A'}$ be a stochastic game graph whose vertex snippet $\mathscr{H}_{A,A'}$ is almost-sure failing under fairness. Then, for all $v \in V^{\mathscr{G}}_{\mathsf{R}} (= V^{\mathscr{H}}_{\mathsf{R}})$,*

$$\inf_{\pi_{\mathsf{R}} \in \Pi^{f}_{\mathsf{R},\mathscr{G}}} \sup_{\pi_{\mathsf{V}} \in \Pi_{\mathsf{V},\mathscr{G}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v}[f_{\mathsf{m}}] = \inf_{\pi_{\mathsf{R}} \in \Pi^{MDf}_{\mathsf{R},\mathscr{H}}} \sup_{\pi_{\mathsf{V}} \in \Pi^{MD}_{\mathsf{V},\mathscr{H}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{H},v}[f_{\mathsf{m}}]$$
$$= \sup_{\pi_{\mathsf{V}} \in \Pi^{MD}_{\mathsf{V},\mathscr{H}}} \inf_{\pi_{\mathsf{R}} \in \Pi^{MDf}_{\mathsf{R},\mathscr{H}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{H},v}[f_{\mathsf{m}}] = \sup_{\pi_{\mathsf{V}} \in \Pi_{\mathsf{V},\mathscr{G}}} \inf_{\pi_{\mathsf{R}} \in \Pi^{f}_{\mathsf{R},\mathscr{G}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v}[f_{\mathsf{m}}].$$

*Proof.* We first recall that the almost-sure failing under fairness property is equivalent to the stopping under fairness property in [9]. That is why we can safely apply the results from [9] on $\mathscr{H}_{A,A'}$ in the calculations below.

$$\begin{aligned}
\inf_{\pi_{\mathsf{R}} \in \Pi^{f}_{\mathsf{R},\mathscr{G}}} \sup_{\pi_{\mathsf{V}} \in \Pi_{\mathsf{V},\mathscr{G}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v}[f_{\mathsf{m}}] &\leq \inf_{\pi_{\mathsf{R}} \in \Pi^{MDf}_{\mathsf{R},\mathscr{G}}} \sup_{\pi_{\mathsf{V}} \in \Pi_{\mathsf{V},\mathscr{G}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v}[f_{\mathsf{m}}] && (\Pi^{MDf}_{\mathsf{R},\mathscr{G}} \subseteq \Pi^{f}_{\mathsf{R},\mathscr{G}}) \quad (\star)\\
&= \inf_{\pi_{\mathsf{R}} \in \Pi^{MDf}_{\mathsf{R},\mathscr{G}}} \sup_{\pi_{\mathsf{V}} \in \Pi^{S}_{\mathsf{V},\mathscr{G}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v}[f_{\mathsf{m}}] && \text{(by Lemma 1)}\\
&= \inf_{\pi_{\mathsf{R}} \in \Pi^{MDf}_{\mathsf{R},\mathscr{G}}} \sup_{\pi_{\mathsf{V}} \in \Pi^{XS}_{\mathsf{V},\mathscr{G}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v}[f_{\mathsf{m}}] && \text{(by Lemma 2)}\\
&= \inf_{\pi_{\mathsf{R}} \in \Pi^{MDf}_{\mathsf{R},\mathscr{H}}} \sup_{\pi_{\mathsf{V}} \in \Pi^{S}_{\mathsf{V},\mathscr{H}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{H},v}[f_{\mathsf{m}}] && \text{(by Prop. 1.1)}\\
&\leq \inf_{\pi_{\mathsf{R}} \in \Pi^{MDf}_{\mathsf{R},\mathscr{H}}} \sup_{\pi_{\mathsf{V}} \in \Pi^{MD}_{\mathsf{V},\mathscr{H}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{H},v}[f_{\mathsf{m}}] && \text{(by [9, Thm. 5])} \quad (\star)\\
&= \sup_{\pi_{\mathsf{V}} \in \Pi^{MD}_{\mathsf{V},\mathscr{H}}} \inf_{\pi_{\mathsf{R}} \in \Pi^{MDf}_{\mathsf{R},\mathscr{H}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{H},v}[f_{\mathsf{m}}] && \text{(by [9, Thm. 5])} \quad (\star)\\
&= \sup_{\pi_{\mathsf{V}} \in \Pi^{MD}_{\mathsf{V},\mathscr{H}}} \inf_{\pi_{\mathsf{R}} \in \Pi^{f}_{\mathsf{R},\mathscr{H}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{H},v}[f_{\mathsf{m}}] && \text{(by [9, Lemma 6])}\\
&= \sup_{\pi_{\mathsf{V}} \in \Pi^{XMD}_{\mathsf{V},\mathscr{G}}} \inf_{\pi_{\mathsf{R}} \in \Pi^{f}_{\mathsf{R},\mathscr{G}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v}[f_{\mathsf{m}}] && \text{(by Prop. 1.2)}\\
&\leq \sup_{\pi_{\mathsf{V}} \in \Pi_{\mathsf{V},\mathscr{G}}} \inf_{\pi_{\mathsf{R}} \in \Pi^{f}_{\mathsf{R},\mathscr{G}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v}[f_{\mathsf{m}}] && (\Pi^{XMD}_{\mathsf{V},\mathscr{G}} \subseteq \Pi_{\mathsf{V},\mathscr{G}}) \quad (\star)\\
&\leq \inf_{\pi_{\mathsf{R}} \in \Pi^{f}_{\mathsf{R},\mathscr{G}}} \sup_{\pi_{\mathsf{V}} \in \Pi_{\mathsf{V},\mathscr{G}}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v}[f_{\mathsf{m}}] && \text{(prop. inf/sup)}
\end{aligned}$$

Formulas marked with $(\star)$ are those in the statement of the theorem and, because the first and last formulas are the same, all of them are equal. □

Theorem 3 guarantees that the stochastic masking game can be solved through its finite vertex snippet using the algorithm proposed in [9]. The next theorem uses this fact to provide a set of Bellman equations based on the symbolic game graph whose greatest fixpoint solution is the solution of the original stochastic masking game.

**Theorem 4.** *Let $\mathscr{G}_{A,A'}$ be a stochastic masking game graph whose vertex snippet is almost-sure failing under fairness and let* $\mathsf{m}$ *be a milestone for $A'$. Let $\mathscr{SG}_{A,A'}$ be the corresponding symbolic game graph. Let $\nu\Gamma$ be the greatest fixpoint of the functional $\Gamma$ defined, for all $v \in V^{\mathscr{SG}}$, as follows:*

$$\Gamma(f)(v) = \begin{cases} \min\left(\mathbf{U}, \max_{w \in \mathbb{V}(\mathbb{C}(v[3],v[4]))} \sum_{v' \in Post(v)} w(v'[0],v'[2]) f(v')\right) & \text{if } v \in V^{\mathscr{SG}}_{\mathsf{P}} \\ \min\left(\mathbf{U}, r^{\mathscr{SG}}_{\mathsf{m}}(v) + \max\{f(v') \mid v' \in Post(v)\}\right) & \text{if } v \in V^{\mathscr{SG}}_{\mathsf{V}} \\ \min\left(\mathbf{U}, \min\{f(v') \mid v' \in Post(v)\}\right) & \text{if } v \in V^{\mathscr{SG}}_{\mathsf{R}} \setminus \{v_{err}\} \\ 0 & \text{if } v = v_{err} \end{cases}$$

*where $v[i]$ is the $i$-th coordinate of $v$ ($i \geq 0$), $r_{\mathsf{m}}^{\mathscr{SG}}(v[0],v[1],v[2],v[3],v[4],v[6]) = r_{\mathsf{m}}^{\mathscr{G}}(v)$ for every $v \in V^{\mathscr{G}}$, and $\mathbf{U} \in \mathbb{R}$ such that $\mathbf{U} \geq \inf_{\pi_{\mathsf{R}} \in \Pi_{\mathsf{R}}^{MDf}} \sup_{\pi_{\mathsf{V}} \in \Pi_{\mathsf{V}}^{MD}} \mathbb{E}^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G}_{A,A'},v}[f_{\mathsf{m}}]$, for every $v \in V^{\mathscr{G}}$. Then, the value of the game $\mathscr{G}_{A,A'}$ at its initial state is equal to $\nu\Gamma(v_0^{\mathscr{SG}})$.*

Constant $\mathbf{U}$ is an upper bound needed so Knaster-Tarski applies on the complete lattice $[0,\mathbf{U}]^V$ [9].

Notice that Theorems 3 and 4 only require $\mathscr{H}_{A,A'}$ to be almost-sure failing under fairness, and if $\mathscr{G}_{A,A'}$ is almost-sure failing under fairness, necessarily so is $\mathscr{H}_{A,A'}$, which makes the theorems stronger. Nonetheless, one would expect that also if $\mathscr{H}_{A,A'}$ is almost-sure failing under fairness, so is $\mathscr{G}_{A,A'}$. That is, we would like that $\inf_{\pi_{\mathsf{V}} \in \Pi_{\mathsf{V}}, \pi_{\mathsf{R}} \in \Pi_{\mathsf{R}}^f} Prob^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v_0^{\mathscr{G}}}(\Diamond v_{\text{err}}) = 1$ if and only if $\inf_{\pi_{\mathsf{V}} \in \Pi_{\mathsf{V}}, \pi_{\mathsf{R}} \in \Pi_{\mathsf{R}}^f} Prob^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{H},v_0^{\mathscr{H}}}(\Diamond v_{\text{err}}) = 1$. Unfortunately we were not able to prove this equivalence, and the most we know (thanks to variants of Lemmas 1 and 2) is that $\inf_{\pi_{\mathsf{V}} \in \Pi_{\mathsf{V}}, \pi_{\mathsf{R}} \in \Pi_{\mathsf{R}}^f} Prob^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{H},v_0^{\mathscr{H}}}(\Diamond v_{\text{err}}) = 1$ implies both $\inf_{\pi_{\mathsf{V}} \in \Pi_{\mathsf{V}}, \pi_{\mathsf{R}} \in \Pi_{\mathsf{R}}^{Sf}} Prob^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v_0^{\mathscr{G}}}(\Diamond v_{\text{err}}) = 1$ and $\inf_{\pi_{\mathsf{V}} \in \Pi_{\mathsf{V}}^S, \pi_{\mathsf{R}} \in \Pi_{\mathsf{R}}^f} Prob^{\pi_{\mathsf{V}},\pi_{\mathsf{R}}}_{\mathscr{G},v_0^{\mathscr{G}}}(\Diamond v_{\text{err}}) = 1$, that is, at least one of the set of strategies needs to be restricted to the semi-Markov ones.

Since $\mathscr{H}_{A,A'}$ is finite, it can be checked whether it is almost-sure failing under fairness by using directly the algorithm proposed in [9, Theorem 3]. However, we could alternatively check it avoiding the explosion introduced by the vertex couplings through the symbolic game graph. Thus, we define the predecessor sets in $\mathscr{SG}_{A,A'}$ for a given set $C$ of symbolic vertices, as follows:

$$\begin{aligned}
\exists Pre_f^{\mathscr{SG}}(C) &= \{v \in V^{\mathscr{SG}} \mid Post(v) \cap C \neq \emptyset\} \\
\forall Pre_f^{\mathscr{SG}}(C) &= \{v \in V_{\mathsf{V}}^{\mathscr{SG}} \mid Post(v) \subseteq C\} \cup \{v \in V_{\mathsf{R}}^{\mathscr{SG}} \mid Post(v) \cap C \neq \emptyset\} \\
&\quad \cup \{v \in V_{\mathsf{P}}^{\mathscr{SG}} \mid Eq(v,C) \text{ has no solution } \}
\end{aligned}$$

$\exists Pre_f^{\mathscr{SG}}(C)$ collects all vertices $v$ for which there is a coupling that leads to a vertex $v'$ in $C$, and do so by simply using the edge $E^{\mathscr{SG}}$ (through $Post$) even for the probabilistic vertices. The definition of $\forall Pre_f^{\mathscr{SG}}(C)$ is more assorted. The first set collects all the Verifier vertices $v$ that inevitably lead to $C$. The second set collects all Refuter vertices $v$ that leads to some state in $C$ (since the Refuter is fair, any successor of $v$ will eventually be taken). The last set collects all probabilistic vertices $v$ for which there is no coupling "avoiding" $C$. This is encoded by checking that $Eq(v,C)$ cannot be solved, since a coupling solving $Eq(v,C)$ defines a probabilistic transition that avoids $C$ with probability 1.

The next theorem provides an algorithm to check whether a vertex snippet is almost-sure failing under fairness using $\exists Pre_f^{\mathscr{SG}}$ and $\forall Pre_f^{\mathscr{SG}}$.

**Theorem 5.** *The vertex snippet $\mathscr{H}_{A,A'}$ of the stochastic masking game $\mathscr{G}_{A,A'}$ is almost-sure failing under fairness if and only if $v_0^{\mathscr{SG}} \in V^{\mathscr{SG}} \setminus \exists Pre_f^{\mathscr{SG}*}(V^{\mathscr{SG}} \setminus \forall Pre_f^{\mathscr{SG}*}(\{v_{err}\}))$, where $v_0^{\mathscr{SG}}$ is the initial state of $\mathscr{SG}_{A,A'}$ (the symbolic version of $\mathscr{G}_{A,A'}$) and $V^{\mathscr{SG}}$ is the sets of vertices of $\mathscr{SG}_{A,A'}$.*

As $Eq(v,C)$ can be computed in polynomial time, so do $\exists Pre_f^{\mathscr{SG}}(C)$ and $\forall Pre_f^{\mathscr{SG}}(C)$. As a consequence, the problem of deciding whether a vertex snippet $\mathscr{H}_{A,A'}$ is almost-sure failing under fairness is polynomial on the sizes of $A$ and $A'$.

# 5 Related Work

Since our metric is a bisimulation-based notion aimed at quantifying how robust a masking fault tolerant algorithm is, the idea of approximate bisimulation immediately shows up. In this category it is worth mentioning $\varepsilon$-bisimulations [21, 16], in which related states that imitate each other do not differ more than an $\varepsilon \in [0,1]$ on the probabilistic value. Therefore $\varepsilon$-bisimulations are not able to accumulate the difference produced in each step. So, these relations cannot measure to what extent faults can be tolerated

over time. The principle of (1-bounded) bisimulation metrics [17, 7] is different as they aim to quantify the similarity of whole models rather than single steps. Nonetheless, if the models inevitably differ (as it is the case of almost-sure failing systems) the metric always equals 1 (maximum difference), which again cannot measure how long faults are tolerated. Instead, bisimulation metrics with discount [17, 7] do give an idea of robustness since the discount factor inversely weights how distant in a trace the difference between the models is eventually witnessed. However, these metrics only provide a relative value (smaller values mean more robust) and cannot focus on particular events as our metric does. In any case, all these notions have been characterized by games which served as a base for algorithmic solutions [16, 7, 3, 33]. In [16] a *non-stochastic* game for $\varepsilon$-bisimulation is provided where each round is divided in five steps in which both Refuter and Verifier alternate twice. Therein the difference is quantified independently in each step, so it is easy to avoid the use of couplings. Instead, the stochastic games for bisimulation metrics [7] are very much similar to ours with the difference that the Verifier only chooses a vertex coupling instead of any possible coupling as we do here, and it considers only deterministic memoryless strategies.

In [26] a weak simulation quasimetric is introduced and used to reason about the evolution of *gossip protocols* to compare protocols with similar behavior up to a certain tolerance. Though its purpose is close to ours, the quasimetric suffers the same problem as bisimulation metrics returning 1 when comparing protocols with almost-sure failing implementations.

Metrics like *Mean-Time To Failure* (MTTF) [29] are normally used. However, our framework is more general than such metrics since it is not limited to count time units as other events may be set as milestones. In addition, the computation of MTTF would normally require the identification of failure states in an ad hoc manner while we do this at a higher level of abstraction.

## 6 Concluding remarks

We presented a relation of masking fault-tolerance between probabilistic transition systems and a corresponding stochastic game characterization. As the game could be infinite, we proposed an alternative finite symbolic representation by means of which the game can be solved in polynomial time. We extended the game with quantitative objectives based on collecting "milestones" thus providing a way to quantify how good an implementation is for masking faults. We proved that the resulting game is determined and can be computed by solving a collection of functional equations. We also provided a polynomial technique to decide whether a game is almost-sure failing under fairness. In this article we focused on the theoretical contribution. We leave as further work the description of the implementation of this idea.

Though it does not affect our result of determinacy nor the algorithmic solution proposed here, it remains open to show whether it holds that whenever the vertex snippet is almost-sure failing under fairness so is the general stochastic masking game. Also, notice that our solution is based on a strong version of bisimulation. A characterization based on probabilistic weak bisimulation would facilitate the application of our approach to complex systems.

## References

[1] Luca de Alfaro, Marco Faella & Mariëlle Stoelinga (2009): *Linear and Branching System Metrics*. *IEEE Trans. Software Eng.* 35(2), pp. 258–273, doi:10.1109/TSE.2008.106.

[2] Giorgio Bacci, Giovanni Bacci, Kim G. Larsen & Radu Mardare (2017): *On-the-Fly Computation of Bisimilarity Distances*. *Log. Methods Comput. Sci.* 13(2), doi:10.23638/LMCS-13(2:13)2017.

[3] Giorgio Bacci, Giovanni Bacci, Kim G. Larsen, Radu Mardare, Qiyi Tang & Franck van Breugel (2019): *Computing Probabilistic Bisimilarity Distances for Probabilistic Automata*. In Wan J. Fokkink & Rob van Glabbeek, editors: *30th International Conference on Concurrency Theory, CONCUR 2019*, *LIPIcs* 140, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 9:1–9:17, doi:10.4230/LIPIcs.CONCUR.2019.9.

[4] Christel Baier & Joost-Pieter Katoen (2008): *Principles of model checking*. MIT Press.

[5] Christel Baier & Marta Z. Kwiatkowska (1998): *Model Checking for a Probabilistic Branching Time Logic with Fairness*. *Distributed Comput.* 11(3), pp. 125–155, doi:10.1007/s004460050046.

[6] Franck van Breugel & James Worrell (2006): *Approximating and computing behavioural distances in probabilistic transition systems*. *Theor. Comput. Sci.* 360(1-3), pp. 373–385, doi:10.1016/j.tcs.2006.05.021.

[7] Franck van Breugel & James Worrell (2014): *The Complexity of Computing a Bisimilarity Pseudometric on Probabilistic Automata*. In Franck van Breugel, Elham Kashefi, Catuscia Palamidessi & Jan Rutten, editors: *Horizons of the Mind. A Tribute to Prakash Panangaden - Essays Dedicated to Prakash Panangaden on the Occasion of His 60th Birthday*, *Lecture Notes in Computer Science* 8464, Springer, pp. 191–213, doi:10.1007/978-3-319-06880-0_10.

[8] Pablo F. Castro, Pedro R. D'Argenio, Ramiro Demasi & Luciano Putruele (2019): *Measuring Masking Fault-Tolerance*. In Tomás Vojnar & Lijun Zhang, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Part II*, *Lecture Notes in Computer Science* 11428, Springer, pp. 375–392, doi:10.1007/978-3-030-17465-1_21.

[9] Pablo F. Castro, Pedro R. D'Argenio, Ramiro Demasi & Luciano Putruele (2022): *Playing Against Fair Adversaries in Stochastic Games with Total Rewards*. In Sharon Shoham & Yakir Vizel, editors: *Computer Aided Verification - 34th International Conference, CAV 2022, Part II*, *Lecture Notes in Computer Science* 13372, Springer, pp. 48–69, doi:10.1007/978-3-031-13188-2_3.

[10] Pavol Cerný, Thomas A. Henzinger & Arjun Radhakrishna (2012): *Simulation distances*. *Theor. Comput. Sci.* 413(1), pp. 21–35, doi:10.1016/j.tcs.2011.08.002.

[11] Krishnendu Chatterjee & Thomas A. Henzinger (2008): *Value Iteration*. In Orna Grumberg & Helmut Veith, editors: *25 Years of Model Checking - History, Achievements, Perspectives*, *Lecture Notes in Computer Science* 5000, Springer, pp. 107–138, doi:10.1007/978-3-540-69850-0_7.

[12] Krishnendu Chatterjee & Thomas A. Henzinger (2012): *A survey of stochastic ω-regular games*. *J. Comput. Syst. Sci.* 78(2), pp. 394–413, doi:10.1016/j.jcss.2011.05.002.

[13] Anne Condon (1990): *On Algorithms for Simple Stochastic Games*. In Jin-Yi Cai, editor: *Advances In Computational Complexity Theory, Proceedings of a DIMACS Workshop, DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 13, DIMACS/AMS, pp. 51–71, doi:10.1090/dimacs/013/04.

[14] Anne Condon (1992): *The Complexity of Stochastic Games*. *Inf. Comput.* 96(2), pp. 203–224, doi:10.1016/0890-5401(92)90048-K.

[15] Josée Desharnais, Vineet Gupta, Radha Jagadeesan & Prakash Panangaden (2004): *Metrics for labelled Markov processes*. *Theor. Comput. Sci.* 318(3), pp. 323–354, doi:10.1016/j.tcs.2003.09.013.

[16] Josée Desharnais, Radha Jagadeesan, Vineet Gupta & Prakash Panangaden (2002): *The Metric Analogue of Weak Bisimulation for Probabilistic Processes*. In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, IEEE Computer Society, pp. 413–422, doi:10.1109/LICS.2002.1029849.

[17] Josée Desharnais, François Laviolette & Mathieu Tracol (2008): *Approximate Analysis of Probabilistic Processes: Logic, Simulation and Games*. In: *Fifth International Conference on the Quantitative Evaluaiton of Systems (QEST 2008), 14-17 September 2008, Saint-Malo, France*, IEEE Computer Society, pp. 264–273, doi:10.1109/QEST.2008.42.

[18] Josée Desharnais, François Laviolette & Amélie Turgeon (2011): *A logical duality for underspecified probabilistic systems*. *Inf. Comput.* 209(5), pp. 850–871, doi:10.1016/j.ic.2010.12.005.

[19] Jerzy Filar & Koos Vrieze (1996): *Competitive Markov Decision Processes*. Springer-Verlag, Berlin, Heidelberg, doi:10.1007/978-1-4612-4054-9.

[20] Felix C. Gärtner (1999): *Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments*. *ACM Comput. Surv.* 31(1), pp. 1–26, doi:10.1145/311531.311532.

[21] Alessandro Giacalone, Chi-Chang Jou & Scott A. Smolka (1990): *Algebraic Reasoning for Probabilistic Concurrent Systems*. In Manfred Broy & Cliff B. Jones, editors: *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, North-Holland, pp. 443–458.

[22] Thomas A. Henzinger (2013): *Quantitative reactive modeling and verification*. *Comput. Sci. Res. Dev.* 28(4), pp. 331–344, doi:10.1007/s00450-013-0251-7.

[23] Thomas A. Henzinger, Rupak Majumdar & Vinayak S. Prabhu (2005): *Quantifying Similarities Between Timed Systems*. In Paul Pettersson & Wang Yi, editors: *Formal Modeling and Analysis of Timed Systems, Third International Conference, FORMATS 2005*, *Lecture Notes in Computer Science* 3829, Springer, pp. 226–241, doi:10.1007/11603009_18.

[24] Edon Kelmendi, Julia Krämer, Jan Kretínský & Maximilian Weininger (2018): *Value Iteration for Simple Stochastic Games: Stopping Criterion and Learning Algorithm*. In Hana Chockler & Georg Weissenbacher, editors: *Computer Aided Verification - 30th International Conference, CAV 2018, Proceedings, Part I*, *Lecture Notes in Computer Science* 10981, Springer, pp. 623–642, doi:10.1007/978-3-319-96145-3_36.

[25] Marta Z. Kwiatkowska, Gethin Norman & David Parker (2011): *PRISM 4.0: Verification of Probabilistic Real-Time Systems*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *Computer Aided Verification - 23rd International Conference, CAV 2011*, *Lecture Notes in Computer Science* 6806, Springer, pp. 585–591, doi:10.1007/978-3-642-22110-1_47.

[26] Ruggero Lanotte, Massimo Merro & Simone Tini (2017): *Weak Simulation Quasimetric in a Gossip Scenario*. In Ahmed Bouajjani & Alexandra Silva, editors: *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017*, *Lecture Notes in Computer Science* 10321, Springer, pp. 139–155, doi:10.1007/978-3-319-60225-7_10.

[27] Kim G. Larsen, Uli Fahrenberg & Claus R. Thrane (2011): *Metrics for weighted transition systems: Axiomatization and complexity*. *Theor. Comput. Sci.* 412(28), pp. 3358–3369, doi:10.1016/j.tcs.2011.04.003.

[28] Kim G. Larsen & Arne Skou (1991): *Bisimulation through Probabilistic Testing*. *Inf. Comput.* 94(1), pp. 1–28, doi:10.1016/0890-5401(91)90030-6.

[29] Jens Lienig & Hans Bruemmer (2017): *Fundamentals of Electronic Systems Design*, chapter Reliability Analysis. Springer International Publishing, doi:10.1007/978-3-319-55840-0.

[30] Roberto Segala (1995): *Modeling and verification of randomized distributed real-time systems*. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA.

[31] Colin Stirling (1998): *The Joys of Bisimulation*. In Lubos Brim, Jozef Gruska & Jirí Zlatuska, editors: *Mathematical Foundations of Computer Science 1998, 23rd International Symposium, MFCS'98*, *Lecture Notes in Computer Science* 1450, Springer, pp. 142–151, doi:10.1007/BFb0055763.

[32] Qiyi Tang & Franck van Breugel (2018): *Deciding Probabilistic Bisimilarity Distance One for Labelled Markov Chains*. In Hana Chockler & Georg Weissenbacher, editors: *Computer Aided Verification - 30th International Conference, CAV 2018, Part I*, *Lecture Notes in Computer Science* 10981, Springer, pp. 681–699, doi:10.1007/978-3-319-96145-3_39.

[33] Qiyi Tang & Franck van Breugel (2020): *Deciding probabilistic bisimilarity distance one for probabilistic automata*. *J. Comput. Syst. Sci.* 111, pp. 57–84, doi:10.1016/j.jcss.2020.02.003.

[34] Claus R. Thrane, Uli Fahrenberg & Kim G. Larsen (2010): *Quantitative analysis of weighted transition systems*. *J. Log. Algebraic Methods Program.* 79(7), pp. 689–703, doi:10.1016/j.jlap.2010.07.010.

# CRIL: A Concurrent Reversible Intermediate Language

Shunya Oguchi Shoji Yuen
Graduate School of Informatics, Nagoya University
Furo-cho, Chikusa-ku, Nagoya 464-8601, Japan
{oguchi321,yuen}@sqlab.jp

We present a reversible intermediate language with concurrency for translating a high-level concurrent programming language to another lower-level concurrent programming language, keeping reversibility. Intermediate languages are commonly used in compiling a source program to an object code program closer to the machine code, where an intermediate language enables behavioral analysis and optimization to be decomposed in steps. We propose CRIL (Concurrent Reversible Intermediate Language) as an extension of RIL used by Mogensen for a functional reversible language, incorporating a multi-thread process invocation and the synchronization primitives based on the P-V operations. We show that the operational semantics of CRIL enjoy the properties of reversibility, including the causal safety and causal liveness proposed by Lanese et al., checking the axiomatic properties. The operational semantics is defined by composing the bidirectional control flow with the dependency information on updating the memory, called *annotation DAG*. We show a simple example of 'airline ticketing' to illustrate how CRIL preserves the causality for reversibility in imperative programs with concurrency.

## 1 Introduction

Reversible programming languages have been proposed to describe reversible computation where the control flows both forward and backward [24, 4, 23, 6]. They directly describe reversible computation and develop new aspects of software development since reversibility holds all information at any point of execution. In forward-only execution, the computation can overwrite the part of its intermediate history unless it is used in the following computation for efficiency. In analyzing the behavior, such as debugging, it is common to replay the execution to the point in focus to recreate the lost part of history. For a concurrent program, replaying the execution is usually difficult since updating shared resources among multiple control threads depends on the runtime environment.

Intermediate languages mediate the translation from the source language to a low-level machine language for execution. Step-by-step translation via intermediate languages is a common technique for optimization in compilers. The intermediate language in LLVM [14] is often used as a behavioral model for program analysis.

Mogensen uses RIL [16] as an intermediate language with reversibility for a functional reversible language in the memory usage analysis. RSSA [17] based on RIL is used for compiling and optimizing Janus programs [9, 3]. Reversibility with concurrency has been studied in process calculi [2, 20, 11, 10], in event structures [18, 19, 21, 15] and recently in programming languages such as Erlang [12] and a simple imperative programming language [6, 8].

We propose a reversible intermediate language CRIL by extending RIL. CRIL extends RIL by allowing multiple blocks to run concurrently and the synchronization primitive based on the P-V operations. In CRIL, concurrent blocks interact with each other via shared variables. To establish the reversibility for concurrent programs, the causality among shared variables has to be preserved. Unlike sequential

G. Caltais and C. A. Mezzina (Eds): Combined Workshop on Expressiveness in Concurrency and Structural Operational Semantics (EXPRESS/SOS 2023).
EPTCS 387, 2023, pp. 149–167, doi:10.4204/EPTCS.387.11

reversible programs, even if one step of a program is reversible, the whole program is not reversible in general since shared variables may not be reversed correctly.

To make a program of CRIL reversible, we give the operational semantics as the labeled transition system, $LTSI_{CRIL}$, as the composition of the operational semantics with one-step reversibility and a data structure called 'annotation DAG'. An annotation DAG accumulates the causality of updating memory in a forward execution and rolls back the causality to control the reversed flow in the backward execution. We show that $LTSI_{CRIL}$ has the basic properties for reversibility proposed in [13]. Using the approach of [13], it is shown that $LTSI_{CRIL}$ enjoys the *Causal Safety* and the *Causal Liveness*, which are important in analyzing CRIL programs compositionally.

By translating a high-level programming language to CRIL, $LTSI_{CRIL}$ works as a virtual machine, and its behavior is guaranteed to be reversible. CRIL enables fine-grained behavioral analysis such as optimization and reversible debugging. In section 4, we present a simple example of airline ticketing given in [5] to enable reversible debugging.

The paper is organized as follows. Section 2 presents the syntax of CRIL and the operational semantics for control flow. Section 3 introduces annotation DAG as a data structure to store the causality of updating memory. We define $LTSI_{CRIL}$ as the operational semantics for CRIL and show the reversibility of $LTSI_{CRIL}$, which is followed by the airline ticketing example in section 4. Section 5 presents concluding remarks.

# 2 CRIL

The syntax of CRIL is defined in figure 1. Following RIL [16], a CRIL program consists of an unordered set of basic blocks. Given a set of labels $\mathscr{L}$, a block has an entry point followed by a block body and an exit point with labels. A block body is either a basic instruction or a call statement.

$$
\begin{array}{rcl}
Pg & ::= & b^* \\
b & ::= & instb \mid callb \\
instb & ::= & entry\ inst\ exit \\
entry & ::= & l \texttt{ <- } \mid l\texttt{;}l \texttt{ <- } e \mid \texttt{begin } l \\
exit & ::= & \texttt{-> } l \mid e \texttt{ -> } l\texttt{;}l \mid \texttt{end } l \\
inst & ::= & left \oplus\texttt{= } e \mid left \texttt{ <-> } left \\
 & & \mid \texttt{V } x \mid \texttt{P } x \mid \texttt{assert } e \mid \texttt{skip} \\
callb & ::= & l \texttt{ <- call } l\texttt{(, }l\texttt{)}^* \texttt{ -> } l \\
e & ::= & right \odot right \mid \texttt{!}right \\
left & ::= & x \mid \texttt{M[}x\texttt{]} \\
right & ::= & k \mid left \\
\oplus & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{\^{}} \\
\odot & ::= & \oplus \mid \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \\
 & & \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||}
\end{array}
$$

Figure 1: The syntax of CRIL

## 2.1 Basic block

We assume all references to variables have a global scope and there exists a heap memory `M` indexed by integers, where `M[`$x$`]` denotes the $x$-th element in `M`. An expression $e$ is either an arithmetic expression or a boolean expression with the usual operators `+`, `-`, `^`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `&&`, `||`, `!` of the C language, where `^` is the bitwise exclusive OR operation. The boolean operators and logical connectives treat 0 as false and any non-0 value as true. An expression can contain integer constants, which are denoted by $k$.

**Entry/exit point** An entry/exit point of a basic block is the following forms:

| Entry point | | Exit point | |
|---|---|---|---|
| (1) | $l$ `<-` | (1') | `->` $l$ |
| (2) | $l_1\texttt{;}l_2$ `<-` $e$ | (2') | $e$ `->` $l_1\texttt{;}l_2$ |
| (3) | `begin` $l$ | (3') | `end` $l$ |

where $l, l_1, l_2 \in \mathscr{L}$. We write $\mathsf{entry}(b)$ for the entry point of a basic block $b$, and $\mathsf{exit}(b)$ for the exit point of a basic block $b$.

The informal meaning of each item is explained as follows:

(1) and (1'): $l$ `<-` receives the control at $l$ unconditionally in a forward execution. In a backward execution, it sends the control to the block that receives the control at $l$. `->` $l$ dually works in the reversed way of $l$ `<-`.

(2) and (2'): $l_1;l_2$ `<-` $e$ receives the control at $l_1$ when $e$ is evaluated to a non-0 value and at $l_2$ otherwise in a forward execution. In a backward execution, it returns the control to the block that receives the control at $l_1$ when $e$ is evaluated to non-0 and at $l_2$ otherwise. $e$ `->` $l_1;l_2$ dually works in the reversed way of $l_1;l_2$ `<-` $e$.

(3) and (3'): `begin` $l$ receives the control from the call statement labeled by $l$ in a forward execution. In a backward execution, it returns the control to the statement labeled by $l$. `end` $l$ dually works in the reversed way of `end` $l$.

A basic block is either an instruction block or a call statement.

**Instruction block** Basic instruction is in the forms:

(1) $left \oplus$`=` $e$ (3) `V` $x$ (5) `assert` $e$
(2) $left_1$ `<->` $left_2$ (4) `P` $x$ (6) `skip`

We write $\mathsf{inst}(b)$ for the basic instruction in $b$. The informal semantics is explained as follows:

(1): $left \oplus$`=` $e$ is an *update* statement where *left* is a left-value, and $\oplus \in \{$`+`, `-`, `^`$\}$. *left* is relatively updated by $e$ in that `+=`, `-=`, and `^=` with the same semantics as in the C language. If $left = x$, $x$ must not appear in $e$. If $left =$ `M[`$x$`]`, heap references must not appear in $e$.

(2): $left_1$ `<->` $left_2$ is an *exchange* where $left_1$ and $left_2$ are left-values. It swaps the values specified by $left_1$ and $left_2$. The same variable must not appear on both sides of `<->`.

(3) and (4): `V` $x$ and `P` $x$ are the P and V operations for synchronization, which correspond to those commonly used in operating systems. We assume variables in `P` and `V` instruction only appear as the parameters of `P` and `V`. In a forward execution, `V` $x$ is defined when $x$ is 0 and terminates when $x$ is 1 and `P` $x$ is defined when $x$ is 1 and terminates when $x$ is 0. In a backward execution, `V` $x$ and `P` $x$ work as `P` $x$ and `V` $x$ of the forward execution respectively.

(5): `assert` $e$ aborts the execution if $e$ evaluates to 0, and does nothing otherwise.

(6): `skip` does nothing in either direction.

We call $\mathscr{R} = Vars \cup \{\mathtt{M}\}$ *memory resources*. Let $\mathsf{Var}(E)$ be the set of memory resource references appearing in $E$, where $E$ is one of *entry*, *exit*, or *inst* in the grammar of figure 1. For example, $\mathsf{Var}($`z-=M[x]+y`$) = \{\mathtt{M}, \mathtt{x}, \mathtt{y}, \mathtt{z}\}$. $\mathsf{read}(b)$ is the memory resources that $b$ uses, and $\mathsf{write}(b)$ is the memory resources that $b$ updates.

$$\mathsf{read}(b) = \mathsf{Var}(\mathsf{entry}(b)) \cup \mathsf{Var}(\mathsf{inst}(b)) \cup \mathsf{Var}(\mathsf{exit}(b))$$

$$\mathsf{write}(b) = \begin{cases} \{x\} & \text{If } \mathsf{inst}(b) = x \oplus \texttt{=}\ e \\ \{\mathtt{M}\} & \text{If } \mathsf{inst}(b) = \texttt{M[}x\texttt{]} \oplus \texttt{=}\ e \\ \{x, y\} & \text{If } \mathsf{inst}(b) = x\ \texttt{<->}\ y \\ \{x, \mathtt{M}\} & \text{If } \mathsf{inst}(b) \in \{x\ \texttt{<->}\ \texttt{M[}y\texttt{]}, \texttt{M[}y\texttt{]}\ \texttt{<->}\ x\} \\ \{\mathtt{M}\} & \text{If } \mathsf{inst}(b) = \texttt{M[}x\texttt{]}\ \texttt{<->}\ \texttt{M[}y\texttt{]} \\ \{x\} & \text{If } \mathsf{inst}(b) \in \{\texttt{P}\ x, \texttt{V}\ x\} \\ \varnothing & \text{Otherwise.} \end{cases}$$

**Call statement** A *call statement* is a basic block in the following form:

$$l \texttt{ <- call } l_1 \texttt{ , } \cdots \texttt{ , } l_n \texttt{ -> } l' \ (n \geq 1)$$

When $n = 1$, it behaves as a subroutine call in RIL. If $n \geq 2$, the controls are simultaneously sent to all basic blocks with `begin` $l_i$ in the forward execution, and to all basic blocks with `end` $l_i$ in the backward execution. In a forward execution, `call` $l_1 \texttt{ , } \cdots \texttt{ , } l_n$ terminates when all controls are returned to this block. In a backward execution, it sends the controls to the blocks whose exit points are `end` $l_i$.

As in RIL, `call` appears only in a basic block whose entry and exit parts are unconditional, and not in any `begin` and `end` blocks. CRIL does not have `uncall` for a call statement since `uncall` makes the semantics more complex in that an uncall nested in multiple calls causes the mixture of forward and backward execution for process blocks. An uncall can be implemented as a symmetrical call.

## 2.2 Process

For a basic block $b$, $\mathsf{in}(b)$,$\mathsf{out}(b) \subseteq \mathscr{L}$ are defined as follows:

$$\mathsf{in}(b) = \begin{cases} \{l\} & \text{if } \mathsf{entry}(b) = l \texttt{ <-} \\ \{l_1, l_2\} & \text{if } \mathsf{entry}(b) = l_1; l_2 \texttt{ <- } e \\ \varnothing & \text{if } \mathsf{entry}(b) = \texttt{begin } l \end{cases} \quad \mathsf{out}(b) = \begin{cases} \{l\} & \text{if } \mathsf{exit}(b) = \texttt{ -> } l \\ \{l_1, l_2\} & \text{if } \mathsf{exit}(b) = e \texttt{ -> } l_1; l_2 \\ \varnothing & \text{if } \mathsf{exit}(b) = \texttt{end } l \end{cases}$$

Basic blocks $b_1$ and $b_2$ are connected, written as $b_1 \bowtie b_2$, if $\mathsf{out}(b_1) \cap \mathsf{in}(b_2) \neq \varnothing$ or $\mathsf{in}(b_1) \cap \mathsf{out}(b_2) \neq \varnothing$. A *process block* of $b$ is $\mathsf{PB}(b, Pg) = \{b' \in Pg \mid b' \bowtie^* b\}$, where $*$ stands for reflexive and transitive closure. No basic block is shared among process blocks since they are the equivalence classes of $\bowtie^*$, which is an equivalence relation on basic blocks.

Let $L_1(B) = \bigcup_{b \in B} (\mathsf{in}(b) \cup \mathsf{out}(b))$ and $L_2(B) = \{l \mid \exists b \in B.\ \mathsf{entry}(b) = \texttt{begin } l \vee \mathsf{exit}(b) = \texttt{end } l\}$. A CRIL program $Pg$ is *well-formed* when it satisfies the following conditions:

(1): For all $l \in L_1(Pg)$, there exists a unique pair $(b_1, b_2)$ such that $\mathsf{in}(b_1) \cap \mathsf{out}(b_2) = \{l\}$;

(2): For all $l \in L_2(Pg)$, there exists a unique pair $(b_1, b_2)$ such that $\mathsf{entry}(b_1) = \texttt{begin } l$ and $\mathsf{exit}(b_2) = \texttt{end } l$;

(3): $L_1(Pg) \cap L_2(Pg) = \varnothing$

(4): For all $b \in Pg$, $|L_2(\mathsf{PB}(b, Pg))| = 1$; and

(5): There is a special label $\texttt{main} \in L_2(Pg)$.

The well-formedness ensures that once a control enters into a process block at `begin` $l$, the control may reach only the matching `end` $l$ in the forward execution and vice versa in the backward execution. A process block $\mathsf{PB}(b, Pg)$ is *labeled* by $l$ when it contains the basic block with `begin` $l$.

A call statement `call` $l_1 \texttt{ , } \cdots \texttt{ , } l_n$ sends controls to process blocks labeled by $l_1, \cdots, l_n$. A process block with control is called a *process*. A process is executed by passing the control among the basic blocks in its process block. Since `call` can be recursive, a process may have some subprocesses.

In a forward execution, $l \texttt{ <- call } l_1 \texttt{ , } \cdots \texttt{ , } l_n \texttt{ -> } l'$ receives a control at $l$, forks $n$ processes, and sends the control to $l'$ after merging the processes. In the backward execution, it works in a reversed manner. In the following, we assume a CRIL program is well-formed.

### 2.3 Basic operational semantics

The set of process identifiers PID is $(\mathbb{N}_+)^*$ where $\mathbb{N}_+$ is the set of positive integers. $p \in \mathsf{PID}$ denotes an identifier uniquely assigned to a process. When $p$ executes a process block $\mathsf{PB}(b, Pg)$, we also write $\mathsf{PB}(p)$. If $p$ is labeled by $l$, $\mathsf{PB}(p) = \mathsf{PB}(b, Pg)$ where $\mathsf{entry}(b) = \texttt{begin}\ l$. A special *root* process has the identifier of $\varepsilon$. The runtime invokes the root process and sends the control to a process block labeled by `main` to start an execution of a CRIL program. For a process $p$, $p \cdot i$ is assigned to the $i$-th subprocess invoked by a call statement of process $p$. $\preceq$ is the prefix relation. A process set $PS$ is a set of process identifiers satisfying (1) $\varepsilon \in PS$; (2) $p \in PS$ implies $p' \in PS$ for $p' \preceq p$; and (3) $p \cdot i$ implies $p \cdot j \in PS$ for $j < i$. For a process set $PS$ and a process id $p$, $\mathsf{isleaf}(PS, p)$ holds if for all $p' \in PS$, $p \preceq p'$ imples $p = p'$.

A *process configuration* is $(l, stage)$, where $l \in \mathscr{L}$ and $stage \in \{\texttt{begin}, \texttt{run}, \texttt{end}\}$ are the location of the control in a process block. If $stage = \texttt{begin}$, it is before executing the process block, if $stage = \texttt{run}$, it is executing the process block, and if $stage = \texttt{end}$ it terminated the process block. PC is the set of process configurations.

A *program configuration* is $(Pg, \rho, \sigma, Pr)$, where $Pg$ is the program (which never changes), $\rho : Vars \to \mathbb{Z}$ maps a variable to its value, $\sigma : \mathbb{N} \to \mathbb{Z}$ maps a heap memory address to its value. A *process map* $Pr : \mathsf{PID} \to \mathsf{PC} \cup \{\bot\}$ maps a process to a process configuration. We assume $Pr_{act}$ is a process set where $Pr_{act} = \{p \in \mathsf{PID} | Pr(p) \in \mathsf{PC}\}$. $\mathscr{C}$ is the set of all program configurations.

A transition relation over program configurations

$$(Pg, \rho, \sigma, Pr) \underset{\text{prog}}{\overset{p, Rd, Wt}{\Longrightarrow}} (Pg, \rho', \sigma', Pr')$$

is defined in figure 2. $(Pg, \rho, \sigma, Pr)$ steps forward to $(Pg, \rho', \sigma', Pr')$ by the process $p$ with reading memory resource $Rd$ and updating memory resource $Wt$. And $(Pg, \rho', \sigma', Pr')$ steps backward to $(Pg, \rho, \sigma, Pr)$ in the same way.

We explain the SOS rules in figure 2. **AssVar** and **AssArr** present the update behavior. The exchange behavior is presented by **SwapVarVar**, **SwapVarArr**, **SwapArrVar**, and **SwapArrArr**. **SwapVarArr** and **SwapArrVar** are reversible since $y$ is evaluated to the same value on both sides of $\rightleftharpoons$. **SwapVarVar** and **SwapArrArr** are clearly reversible. **Skip** presents the skip behavior. **Assert** presents the assertion behavior, which stops when $e$ is evaluated to 0.

**V-op** and **P-op** present the behavior of `V` $x$ and `P` $x$ for synchronization by $x$ shared among concurrent processes. In forward execution, `V` $x$ sets $x = 1$ when $x = 0$, and waits otherwise. In backward execution, `V` $x$ sets $x = 0$ when $x = 1$, and waits otherwise. `P` behaves in a symmetrical fashion. By the pair of `V` $x$ and `P` $x$, $x$ can be used as a semaphore to implement the mutual exclusion for both directions of execution.

**Inst** presents the one-step behavior of a basic block. The instruction updates $\rho$ and $\sigma$ and the entry and exit points give the status of the process. The process is running if $stage$ is `run`. the process is at the initial block or at the final block, if $stage$ is `begin` or `end`. The transition label $Rd$ is $\texttt{read}(b)$ and the transition label $Wt$ is $\texttt{write}(b)$.

**CallFork** presents that a call statement forks subprocesses. When $p$ executes a call statement `call` $l_1, \cdots, l_n$ forwards, it forks subprocesses labeled by $l_1, \cdots, l_n$ and $p$ stores the label for returning the controls in $Pr$. Note that the process map is changed to $Pr'$ with subprocesses after forking subprocesses. Since $\mathsf{isleaf}(Pr'_{act}, p)$ does not hold, $p$ does not pass the control to the next block until all the subprocesses are merged. **CallMerge** works dually to **CallFork**. In a forward execution, when all subprocesses reach the `end` stage, all subprocesses are set to inactive and $p$ resumes to pass the control to the next basic block. In a backward execution, **CallFork** behaves as **CallMerge** of forward execution and vice versa for **CallMerge**.

Expressions:

$$\frac{k \text{ is a constant}}{(\rho,\sigma) \rhd k \rightsquigarrow k}\ \textbf{Con} \qquad \frac{}{(\rho[x \mapsto m],\sigma) \rhd x \rightsquigarrow m}\ \textbf{Var} \qquad \frac{}{(\rho[x \mapsto m_1],\sigma[m_1 \mapsto m_2]) \rhd M[x] \rightsquigarrow m_2}\ \textbf{Mem}$$

$$\frac{(\rho,\sigma) \rhd \mathit{right}_1 \rightsquigarrow m_1 \qquad (\rho,\sigma) \rhd \mathit{right}_2 \rightsquigarrow m_2 \qquad m_3 = m_1 \odot m_2}{(\rho,\sigma) \rhd \mathit{right}_1 \odot \mathit{right}_2 \rightsquigarrow m_3}\ \textbf{Exp1}$$

$$\frac{(\rho,\sigma) \rhd \mathit{right} \rightsquigarrow 0}{(\rho,\sigma) \rhd \texttt{!}\mathit{right} \rightsquigarrow 1}\ \textbf{Exp2} \qquad \frac{(\rho,\sigma) \rhd \mathit{right} \rightsquigarrow m \qquad m \neq 0}{(\rho,\sigma) \rhd \texttt{!}\mathit{right} \rightsquigarrow 0}\ \textbf{Exp3}$$

Instructions:

$$\frac{(\rho,\sigma) \rhd e \rightsquigarrow m_3 \qquad m_2 = m_1 \oplus m_3}{x \oplus\texttt{=}\ e \rhd (\rho[x \mapsto m_1],\sigma) \rightleftharpoons (\rho[x \mapsto m_2],\sigma)}\ \textbf{AssVar}$$

$$\frac{(\rho,\sigma) \rhd e \rightsquigarrow m_3 \qquad m_2 = m_1 \oplus m_3}{\texttt{M[}x\texttt{]} \oplus\texttt{=}\ e \rhd (\rho[x \mapsto m_4],\sigma[m_4 \mapsto m_1]) \rightleftharpoons (\rho[x \mapsto m_4],\sigma[m_4 \mapsto m_2])}\ \textbf{AssArr}$$

$$\frac{}{x\ \texttt{<->}\ y \rhd (\rho[x,y \mapsto m_1,m_2],\sigma) \rightleftharpoons (\rho[x,y \mapsto m_2,m_1],\sigma)}\ \textbf{SwapVarVar}$$

$$\frac{}{x\ \texttt{<->}\ \texttt{M[}y\texttt{]} \rhd (\rho[x,y \mapsto m_1,m_3],\sigma[m_3 \mapsto m_2]) \rightleftharpoons (\rho[x,y \mapsto m_2,m_3],\sigma[m_3 \mapsto m_1])}\ \textbf{SwapVarArr}$$

$$\frac{}{\texttt{M[}y\texttt{]}\ \texttt{<->}\ x \rhd (\rho[x,y \mapsto m_1,m_3],\sigma[m_3 \mapsto m_2]) \rightleftharpoons (\rho[x,y \mapsto m_2,m_3],\sigma[m_3 \mapsto m_1])}\ \textbf{SwapArrVar}$$

$$\frac{}{\texttt{M[}x\texttt{]}\ \texttt{<->}\ \texttt{M[}y\texttt{]} \rhd (\rho[x,y \mapsto m_3,m_4],\sigma[m_3,m_4 \mapsto m_1,m_2]) \rightleftharpoons (\rho[x,y \mapsto m_3,m_4],\sigma[m_3,m_4 \mapsto m_2,m_1])}\ \textbf{SwapArrArr}$$

$$\frac{}{\texttt{V}\ x \rhd (\rho[x \mapsto 0],\sigma) \rightleftharpoons (\rho[x \mapsto 1],\sigma)}\ \textbf{V-op} \qquad \frac{}{\texttt{P}\ x \rhd (\rho[x \mapsto 1],\sigma) \rightleftharpoons (\rho[x \mapsto 0],\sigma)}\ \textbf{P-op}$$

$$\frac{}{\texttt{skip} \rhd (\rho,\sigma) \rightleftharpoons (\rho,\sigma)}\ \textbf{Skip} \qquad \frac{(\rho,\sigma) \rhd e \rightsquigarrow m \qquad m \neq 0}{\texttt{assert}\ e \rhd (\rho,\sigma) \rightleftharpoons (\rho,\sigma)}\ \textbf{Assert}$$

Entry and exit points:

$$\frac{}{\texttt{begin}\ l \vdash (\rho,\sigma,l,\texttt{begin})} \qquad \frac{}{l\ \texttt{<-} \vdash (\rho,\sigma,l,\texttt{run})} \qquad \frac{\rho\ \sigma \rhd e \rightsquigarrow 0}{l_1\texttt{;}l_2\ \texttt{<-}\ e \vdash (\rho,\sigma,l_2,\texttt{run})} \qquad \frac{\rho\ \sigma \rhd e \rightsquigarrow m \qquad m \neq 0}{l_1\texttt{;}l_2\ \texttt{<-}\ e \vdash (\rho,\sigma,l_1,\texttt{run})}$$

$$\frac{}{\texttt{end}\ l \dashv (\rho,\sigma,l,\texttt{end})} \qquad \frac{}{\texttt{->}\ l \dashv (\rho,\sigma,l,\texttt{run})} \qquad \frac{\rho\ \sigma \rhd e \rightsquigarrow 0}{e\ \texttt{->}\ l_1\texttt{;}l_2 \dashv (\rho,\sigma,l_2,\texttt{run})} \qquad \frac{\rho\ \sigma \rhd e \rightsquigarrow m \qquad m \neq 0}{e\ \texttt{->}\ l_1\texttt{;}l_2 \dashv (\rho,\sigma,l_1,\texttt{run})}$$

Basic Blocks:

$$\frac{\mathsf{isleaf}(Pr_{act},p) \quad b \in Pg \qquad \mathsf{entry}(b) \vdash (\rho,\sigma,l,\mathit{stage}) \quad \mathsf{inst}(b) \rhd (\rho,\sigma) \rightleftharpoons (\rho',\sigma') \quad \mathsf{exit}(b) \dashv (\rho',\sigma',l',\mathit{stage}')}{(Pg,\rho,\sigma,Pr[p \mapsto (l,\mathit{stage})]) \overset{p,\mathsf{read}(b),\mathsf{write}(b)}{\underset{\text{prog}}{\Longrightarrow}} (Pg,\rho',\sigma',Pr[p \mapsto (l',\mathit{stage}')])}\ \textbf{Inst}$$

$$\frac{\mathsf{isleaf}(Pr_{act},p) \qquad (l'\ \texttt{<-},\texttt{call}\ l_1,\cdots,l_n,\texttt{->}\ l'') \in Pg}{(Pg,\rho,\sigma,Pr[p \mapsto (l',\texttt{run})]) \overset{p,\varnothing,\varnothing}{\underset{\text{prog}}{\Longrightarrow}} (Pg,\rho,\sigma,Pr[p \mapsto (l'',\texttt{run}),p\cdot 1 \mapsto (l_1,\texttt{begin}),\cdots,p\cdot n \mapsto (l_n,\texttt{begin})])}\ \textbf{CallFork}$$

$$\frac{\mathsf{isleaf}(Pr_{act},p) \qquad (l'\ \texttt{<-},\texttt{call}\ l_1,\cdots,l_n,\texttt{->}\ l'') \in Pg}{(Pg,\rho,\sigma,Pr[p \mapsto (l'',\texttt{run}),p\cdot 1 \mapsto (l_1,\texttt{end}),\cdots,p\cdot n \mapsto (l_n,\texttt{end})]) \overset{p,\varnothing,\varnothing}{\underset{\text{prog}}{\Longrightarrow}} (Pg,\rho,\sigma,Pr[p \mapsto (l'',\texttt{run})])}\ \textbf{CallMerge}$$

Figure 2: The basic operational semantics

In a program configuration of CRIL, there is no stack as in RIL to store the return label for subroutine calls. The process map stores the return label, which is not available until $\mathsf{isleaf}(Pr_{act}, p)$ holds, where it checks if the label is on the stack.

Figure3 shows an example of CRIL program $Pg$. There are four process blocks $\{b_1,b_2,b_3\}$,$\{b_4,b_5\}$, $\{b_6\}$, and $\{b_7\}$. A process map assigns $\varepsilon$ to $\{b_1,b_2,b_3\}$. In the following execution, it assigns 1 to $\{b_4,b_5\}$, 2 to $\{b_6\}$, and 3 to $\{b_7\}$.

An example of the transitions for $Pg$ is as follows:

$(Pg, \rho_0, \sigma_0, [\varepsilon \mapsto (\texttt{main}, \texttt{begin})])$

$\xrightleftharpoons[\text{prog}]{\varepsilon,\emptyset,\emptyset} (Pg, \rho_0, \sigma_0, [\varepsilon \mapsto (\texttt{l1}, \texttt{run})])$

$\xrightleftharpoons[\text{prog}]{\varepsilon,\emptyset,\emptyset} (Pg, \rho_0, \sigma_0, \begin{array}{l}[\varepsilon \mapsto (\texttt{l2}, \texttt{run}), 1 \mapsto (\texttt{begin}, \texttt{sub0}),\\ 2 \mapsto (\texttt{sub1}, \texttt{begin}), 3 \mapsto (\texttt{sub2}, \texttt{begin})]\end{array})$

$\xrightleftharpoons[\text{prog}]{1,\{\texttt{x}\},\{\texttt{x}\}} (Pg, \rho_1, \sigma_0, \begin{array}{l}[\varepsilon \mapsto (\texttt{l2}, \texttt{run}), 1 \mapsto (\texttt{l3}, \texttt{run}),\\ 2 \mapsto (\texttt{sub1}, \texttt{begin}), 3 \mapsto (\texttt{sub2}, \texttt{begin})]\end{array})$
where $\rho_1 = \rho_0[\texttt{x} \mapsto 1]$

$\xrightleftharpoons[\text{prog}]{2,\{\texttt{x},\texttt{y}\},\{\texttt{x}\}} (Pg, \rho_2, \sigma_0, \begin{array}{l}[\varepsilon \mapsto (\texttt{l2}, \texttt{run}), 1 \mapsto (\texttt{l3}, \texttt{run}),\\ 2 \mapsto (\texttt{sub1}, \texttt{end}), 3 \mapsto (\texttt{sub2}, \texttt{begin})]\end{array})$
where $\rho_2 = \rho_2[\texttt{y} \mapsto 1]$

$\xrightleftharpoons[\text{prog}]{3,\{\texttt{x},\texttt{z}\},\{\texttt{z}\}} (Pg, \rho_3, \sigma_0, \begin{array}{l}[\varepsilon \mapsto (\texttt{l2}, \texttt{run}), 1 \mapsto (\texttt{l3}, \texttt{run}),\\ 2 \mapsto (\texttt{sub1}, \texttt{end}), 3 \mapsto (\texttt{sub2}, \texttt{end})]\end{array})$
where $\rho_3 = \rho_1[\texttt{z} \mapsto 1]$

$\xrightleftharpoons[\text{prog}]{1,\{\texttt{x}\},\{\texttt{x}\}} (Pg, \rho_4, \sigma_0, \begin{array}{l}[\varepsilon \mapsto (\texttt{l2}, \texttt{run}), 1 \mapsto (\texttt{sub0}, \texttt{end}),\\ 2 \mapsto (\texttt{sub1}, \texttt{end}), 3 \mapsto (\texttt{sub2}, \texttt{end})]\end{array})$
where $\rho_4 = \rho_3[\texttt{x} \mapsto 2]$

$\xrightleftharpoons[\text{prog}]{\varepsilon,\emptyset,\emptyset} (Pg, \rho_4, \sigma_0, [\varepsilon \mapsto (\texttt{l2}, \texttt{run})])$

$\xrightleftharpoons[\text{prog}]{\varepsilon,\emptyset,\emptyset} (Pg, \rho_4, \sigma_0, [\varepsilon \mapsto (\texttt{main}, \texttt{end})])$

```
b1  begin main
    skip
    -> l1
b2  l1 <-
    call sub0, sub1, sub2
    -> l2
b3  l2 <-
    skip
    end main

b4  begin sub0
    x += 1
    -> l3
b5  l3 <-
    x += 1
    end sub0

b6  begin sub1
    y += x
    end sub1

b7  begin sub2
    z += x
    end sub2
```

Figure 3: A CRIL program $Pg$

This forward execution ends with $\texttt{x} = 2, \texttt{y} = 1, \texttt{z} = 1$. The operational semantics show that the computation may be reversed to $(Pg, \rho_0, \sigma_0, [\varepsilon \mapsto (\texttt{main}, \texttt{begin})])$. However, it is possible to reverse to a different configuration such as $\texttt{x} = 0, \texttt{y} = -1, \texttt{z} = -1$ if the call statement is reversed in a different order. Thus, this operational semantics is not reversible. In the next section, we will combine an annotation for the dependency information as DAG to make the basic properties for reversibility as well as Causal Safety and Causal Liveness.

## 3 Reversibility of CRIL

Table 1 (a) shows the transitions of store $\rho$ by the sequence of basic blocks in the forward computation of the example in the previous section. Process $p$ makes the forward (left-to-right) transition of $\xrightleftharpoons[\text{prog}]{p,Rd,Wt}$. The program configuration at the end is $(Pg, [\texttt{x} \mapsto 2, \texttt{y} \mapsto 1, \texttt{z} \mapsto 1], \sigma_0, [\varepsilon \mapsto (\texttt{main}, \texttt{end})]$. The configuration may lead to a different store by the backward (right-to-left) transitions of $\xrightleftharpoons[\text{prog}]{p,Rd,Wt}$ as shown in table 1 (b). Although each step of the operational semantics keeps the local reversibility, it does not preserve the causality of shared memory. The forward step of $\xrightleftharpoons[\text{prog}]{p,Rd,Wt}$ updates $Wt$ reading $Rd$ making the causality from $Rd$ to $Wt$. Our idea is to control processes to keep the causality by observing $Rd$ and $Wt$ being combined with the operational semantics.

| | x | y | z |
|---|---|---|---|
| | 0 | 0 | 0 |
| $b_1 \in \mathsf{PB}(\varepsilon)$ | | | |
| | 0 | 0 | 0 |
| $b_2 \in \mathsf{PB}(\varepsilon)$ | | | |
| | 0 | 0 | 0 |
| $b_4 \in \mathsf{PB}(1)$ | | | |
| | 1 | 0 | 0 |
| $b_6 \in \mathsf{PB}(2)$ | | | |
| | 1 | 1 | 0 |
| $b_7 \in \mathsf{PB}(3)$ | | | |
| | 1 | 1 | 1 |
| $b_5 \in \mathsf{PB}(1)$ | | | |
| | 2 | 1 | 1 |
| $b_2 \in \mathsf{PB}(\varepsilon)$ | | | |
| | 2 | 1 | 1 |
| $b_3 \in \mathsf{PB}(\varepsilon)$ | | | |
| | 2 | 1 | 1 |

(a) A forward Execution

| | x | y | z |
|---|---|---|---|
| | 2 | 1 | 1 |
| $b_3 \in \mathsf{PB}(\varepsilon)$ | | | |
| | 2 | 1 | 1 |
| $b_2 \in \mathsf{PB}(\varepsilon)$ | | | |
| | 2 | 1 | 1 |
| $b_7 \in \mathsf{PB}(3)$ | | | |
| | 2 | −1 | 1 |
| $b_6 \in \mathsf{PB}(2)$ | | | |
| | 2 | −1 | −1 |
| $b_5 \in \mathsf{PB}(1)$ | | | |
| | 1 | −1 | −1 |
| $b_4 \in \mathsf{PB}(1)$ | | | |
| | 0 | −1 | −1 |
| $b_2 \in \mathsf{PB}(\varepsilon)$ | | | |
| | 0 | −1 | −1 |
| $b_1 \in \mathsf{PB}(\varepsilon)$ | | | |
| | 0 | −1 | −1 |

(b) The corresponding backward execution

Table 1: Store changes in executions

| | read | write |
|---|---|---|
| $b_1$ | $\varnothing$ | $\varnothing$ |
| $b_2$ | $\varnothing$ | $\varnothing$ |
| $b_3$ | $\varnothing$ | $\varnothing$ |
| $b_4$ | {x} | {x} |
| $b_5$ | {x} | {x} |
| $b_6$ | {x,y} | {y} |
| $b_7$ | {x,z} | {z} |

Table 2: read and write for basic blocks

Table 2 presents read and write for each basic block. In the backward execution, after reversing $b_3b_2$, **CallMerge** works as a forking of three processes in backward. At this point, $b_5$, $b_6$, and $b_7$ are possible by using the rule backward. Since $\mathsf{write}(b_5) = \{\mathtt{x}\}$ and both $\mathsf{read}(b_6)$ and $\mathsf{read}(b_7)$ contain $x$, the order between $b_5$ and $b_6$, and the order between $b_5$ and $b_6$ affect the causality. We say $b_i$ *conflicts* with $b_j$ where $i \neq j$ if $\mathsf{read}(b_i) \cap \mathsf{write}(b_j) \neq \varnothing$ or $\mathsf{read}(b_j) \cap \mathsf{write}(b_i) \neq \varnothing$. Since $b_6$ and $b_7$ do not conflict with each other, the order between $b_6$ and $b_7$ does not affect the causality. Thus, for the forward execution in table 1 (a), the reversed execution $b_3b_2b_3b_6b_7b_4b_2b_1$ reaches $\rho_0$ as a legitimate reversed computation.

### 3.1 Annotation DAG

We shall present a data structure called 'annotation DAG' (Directed Acyclic Graph) that keeps the conflicting information in forward execution and controls the backward execution by matching the causality, observing the memory $Wt$ updated by reading the memory $Rd$.

**Definition 1.** *An annotation DAG is $A = (V, E_R, E_W)$ satisfying the following conditions:*

1. $V \subseteq (\mathsf{PID} \times \mathbb{N}) \cup \{\bot\}$ *where $\mathbb{N}$ is the set of natural numbers, $\bot \in V$, and if $(p,n) \in V$ then for all $n' \leq n$, $(p,n') \in V$;*
2. $E_R, E_W \subseteq V \times \mathscr{R} \times V$ *where $(v',r,v),(v'',r,v) \in E_R \cup E_W$ implies $v' = v''$;*
3. $E_R \cap E_W = \varnothing$ *and $(V, E_R \uplus E_W)$ is a DAG with the finite set of nodes $V$;*
4. $(v',r,v) \in E_W$ *and $v' \neq \bot$ imply $(v'',r,v') \in E_W$; and*
5. $(v,r,v'),(v,r,v'') \in E_W$ *implies $v' = v''$*

*$\mathscr{A}$ is the set of all annotation DAGs, and $A_{init}$ is $(\{\bot\},\varnothing,\varnothing)$.*

We write $v \xrightarrow{r} v'$ for $(v,r,v') \in E_W$ and $v \overset{r}{\dashrightarrow} v'$ for $(v,r,v') \in E_R$. Condition 5 with conditions 3 and 2 ensures that when $v' \xrightarrow{r} v$, there is a unique sequence of $E_W$ with the label of $r$ from $\bot$ to $v$: $\bot \xrightarrow{r} v_1 \xrightarrow{r} \cdots \xrightarrow{r} v_n = v$. $\mathsf{last}(r, E_W)$ denotes the last node $v$ of such sequence. When $\mathsf{last}(r, E_W) = v \neq \bot$, $v' \xrightarrow{r} v$ for a unique $v'$ and $v \overset{r}{\not\to} v''$ for all $v''$. $\mathsf{last}(r,\varnothing) = \bot$ for all $r \in \mathscr{R}$. Since $V$ is finite, for $(p,n) \in V$

there is the maximum number for process $p$ if such $(p,n)$ exists. Given $V \subseteq \mathsf{PID} \times \mathbb{N} \cup \{\bot\}$, we write $\mathsf{max}_p(V)$ for $max_{(p,n)\in V}\, n$ for some $(p,n) \in V$. $\mathsf{max}_p(V) = -1$ when $(p,n) \notin V$ for all $n$.

**Definition 2.** *For* $A_1, A_2 \in \mathscr{A}$, $A_1 = (V_1, E_{R1}, E_{W1}) \xrightleftharpoons[\text{ann}]{p,Rd,Wt} A_2 = (V_2, E_{R2}, E_{W2})$ *if*

1. $V_2 = V_1 \cup \{v\}$;
2. $E_{R2} = E_{R1} \cup \{\mathsf{newedge}(r, E_{W1}, v) \mid r \in Rd - Wt\}$;*and*
3. $E_{W2} = E_{W1} \cup \{\mathsf{newedge}(r, E_{W1}, v) \mid r \in Wt\}$

*where* $v = (p, \mathsf{max}_p(V_1) + 1)$ *and* $\mathsf{newedge}(r, E_W, v) = (\mathsf{last}(r, E_W), r, v)$.

Given $O \subseteq \mathsf{PID} \times 2^{\mathscr{R}} \times 2^{\mathscr{R}}$, $A \xrightleftharpoons[\text{ann}]{O}{}^{+} A'$ when for some $(p_i, Rd_i, Wt_i) \in O$, there exists a sequence of $A_i$ such that $A_0 \xrightleftharpoons[\text{ann}]{p_1,Rd_1,Wt_1} A_1 \cdots \xrightleftharpoons[\text{ann}]{p_n,Rd_n,Wt_n} A_n = A'$. We write $A \xrightleftharpoons[\text{ann}]{O}{}^{*} A'$ if $A = A'$ or $A \xrightleftharpoons[\text{ann}]{O}{}^{+} A'$.

Intuitively, the forward (left-to-right) relation of $A_1 = (V_1, E_{R1}, E_{W1}) \xrightleftharpoons[\text{ann}]{p,Rd,Wt} A_2$ is to add a fresh node $v$ and edges $\mathsf{newedge}(r, E_{W1}, v)$ for $r \in Rd \cup Wt$ to $A_1$ for process $p$ to execute forwards a basic block $b \in \mathsf{PB}(p)$ with $\mathsf{read}(b) = Rd$ and $\mathsf{write}(b) = Wt$. $v$ presents the new causality created by executing process $p$ in the forward direction. The new node has the causality to update $Wt$ from $(p, \mathsf{max}_p(V_1))$ that presents the newest causality in $A_1$ by $p$ in forward. To update the causality, the edges $\mathsf{newedge}(r, E_{W1}, v)$ for $r \in Wt$ are added to $E_{W1}$. The edges $\mathsf{newedge}(r, E_{W1}, v)$ for $r \in Rd - Wt$ from the newest causality for $r$ at that moment are added to $E_{R1}$ to show that the update for $v$ depends on such $r$.

The backward (right-to-left) relation is to remove a node and edges from $A_2$. The node $v$ to be removed has to be the newest causality of a process and does not depend on other causalities. It is shown that such a node always exists in an annotation DAG in $\{A \mid A_{init} \xrightleftharpoons[\text{ann}]{O}{}^{+} A\}$ as below.

**Proposition 1.** *For* $(V, E_R, E_W) \in \{A \mid A_{init} \xrightleftharpoons[\text{ann}]{}{}^{+} A\}$, *there exists a node* $v \in V$ *such that* $v' \overset{r}{\dashrightarrow} v$ *implies* $v' = \mathsf{last}(r, E_W)$; *and no outgoing edge from* $v$.

Moreover, $\mathscr{A}^O_{comp} = \{A \mid A_{init} \xrightleftharpoons[\text{ann}]{O}{}^{*} A\}$ is closed by $\xrightleftharpoons[\text{ann}]{p,Rd,Wt}$ where $(p, Rd, Wt) \in O$. Obviously, $A \in \mathscr{A}_{comp}$ implies $A' \in \mathscr{A}^O_{comp}$ when $A \xrightleftharpoons[\text{ann}]{p,Rd,Wt} A'$ for some $(p, Rd, Wt) \in O$ by definition.

**Proposition 2.** *For* $A \in \{A' \mid A_{init} \xrightleftharpoons[\text{ann}]{O}{}^{+} A'\}$, *there exists* $A'' \in \mathscr{A}^O_{comp}$ *such that* $A'' \xrightleftharpoons[\text{ann}]{p,Rd,Wt} A$ *with* $(p, Rd, Wt) \in O$.

## 3.2 Operational semantics controlled by Annotation DAG

**Definition 3.** *The* operational semantics controlled by annotation DAG *over program configurations* $(C, A) \xrightleftharpoons{p,Rd,Wt} (C', A')$ *is defined by:*

$$\frac{C \xrightleftharpoons[\text{prog}]{p,Rd,Wt} C' \qquad A \xrightleftharpoons[\text{ann}]{p,Rd,Wt} A'}{(C, A) \xrightleftharpoons{p,Rd,Wt} (C', A')}\ \textbf{ProgAnn}$$

*where* $p \in \mathsf{PID}$ *and* $Rd, Wt \subseteq \mathscr{R}$.

*The* program computation with annotation *is a sequence of* $(C_i, A_i) \xrightleftharpoons{p_i,Rd_i,Wt_i} (C_{i+1}, A_{i+1})$ $(i \geq 0)$ *beginning with* $(C_0, A_0) = (C_{init}, A_{init})$.

We illustrate the behavior controlled by the annotation DAG for the simple example of the previous section. Starting from the initial configuration,$(C_0,A_0) = ((Pg,\rho_0,\sigma_0,[(\varepsilon \mapsto (\texttt{main},\texttt{begin}))]),(\{\bot\},\varnothing,\varnothing))$, it ends up with $(C_8,A_8) = (P_g,[x,y,z \mapsto 2,1,1],\sigma_0,[\varepsilon \mapsto (\texttt{main},\texttt{end})])$.

**Forward accumulation of causality** We present the construction of annotation DAGs as follows:

(F1) After process $\varepsilon$ executes $b_1$ and $b_2$, $A_2 = (\{\bot,(\varepsilon,0),(\varepsilon,1)\},\varnothing,\varnothing)$;

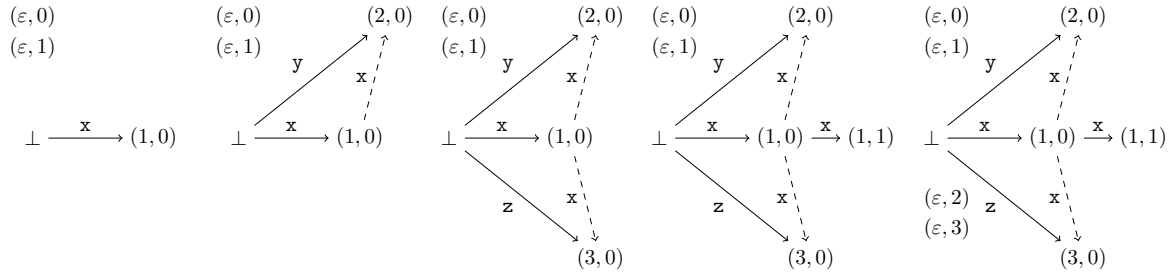
(F2) The call statement in $b_2$ forks three subprocesses. Then, process 1 executes $b_4$, $(1,0)$ is added to $V$ and $\bot \xrightarrow{\texttt{x}} (1,0)$ is added since $\mathsf{read}(b_4) = \mathsf{write}(b_4) = \{\texttt{x}\}$ to make $A_3$, meaning x is updated by the initial x, and the store is updated as $[\texttt{x},\texttt{y},\texttt{z} \mapsto 1,0,0]$.

(F3) Next, process 2 executes $b_6$ where $\mathsf{read}(b_6) = \{\texttt{x},\texttt{y}\}$ and $\mathsf{write}(b_6) = \{\texttt{y}\}$. $\underset{\text{ann}}{\overset{2,\{\texttt{x},\texttt{y}\},\{\texttt{y}\}}{\rightleftharpoons}}$ adds a fresh node $(2,0)$, $\bot \xrightarrow{\texttt{y}} (2,0)$, and $(1,0) \overset{\texttt{x}}{\dashrightarrow} (2,0)$. The causality of $(2,0)$ means $y$ is updated by the initial y and x of $(1,0)$ to make $A_4$.

(F4) Then, process 3 executes $b_7$ where $\mathsf{read}(b_7) = \{\texttt{x},\texttt{z}\}$ and $\mathsf{write}(b_7) = \{\texttt{z}\}$. $\underset{\text{ann}}{\overset{3,\{\texttt{x},\texttt{z}\},\{\texttt{z}\}}{\rightleftharpoons}}$ adds $(3,0)$, $\bot \xrightarrow{\texttt{z}} (3,0)$, and $(1,0) \overset{\texttt{x}}{\dashrightarrow} (3,0)$, to make $A_5$ shown in figure 4 (a), meaning the causality at $(3,0)$ to update the initial z using the initial z and x of $(1,0)$.

(F5) At last, process 1 executes $b_5$ where $\mathsf{read}(b_5) = \mathsf{write}(b_5) = \{\texttt{x}\}$. $\underset{\text{ann}}{\overset{1,\{\texttt{x}\},\{\texttt{x}\}}{\rightleftharpoons}}$ just adds $(1,1)$ and $(1,0) \xrightarrow{\texttt{x}} (1,1)$ to form $A_6$ shown in figure 4 (b), meaning x is updated by x of $(1,0)$.

(F6) No more causality is created after merging the subprocesses. Just the relation adds $(\varepsilon,2)$ and $(\varepsilon,3)$ with no edges to form $A_8$ shown in figure 4 (c).




(a) $A_3$ (b) $A_4$ (c) $A_5$ (d) $A_6$ (e) $A_8$

Figure 4: Annotation DAGs along with forward execution

**Backward rollback of causality** The following is the summary of the corresponding backward execution.

(B1) The removable nodes of $A_8$ are $\{(\varepsilon,3),(1,1)\}$. Here, $C_8$ specifies $\varepsilon$ to remove $(\varepsilon,3)$ followed by removing $(\varepsilon,2)$ back to $(C_6,A_6)$, where $C_6 = (Pg,[\texttt{x},\texttt{y},\texttt{z} \mapsto 2,1,1],\sigma_0,[\varepsilon \mapsto (\texttt{l2},\texttt{run}),1 \mapsto (\texttt{sub0},\texttt{end}),2 \mapsto (\texttt{sub1},\texttt{end}),3 \mapsto (\texttt{sub2},\texttt{end})])$

(B2) $C_6$ may reverse any subprocess, but $A_6$ allows only to remove $(1,1)$ by $\underset{\text{ann}}{\overset{p,Rd,Wt}{\rightleftharpoons}}$ to obtain $A_5$.

(B3) After removing $(1,1)$ and $(1,0) \xrightarrow{\texttt{x}} (1,1)$ from $A_6$, we obtain $A_5$ whose removable nodes are $(2,0)$ and $(3,0)$. $(1,0)$ is not removable since $(1,0)$ has two outgoing edges, although $(1,0) = \mathsf{last}(\texttt{x},E_W)$.

(B4) $C_5$ may reverse either process 2 or process 3, and let process 2 reverse to become $C'_4$. Then, remove $\bot \xrightarrow{y} (2,0)$ and $(1,0) \dashrightarrow^{x} (2,0)$ to obtain $A'_4$ and $[x,y,z \mapsto 1,0,1]$ as the store $\rho$. Note that $(C'_4,A'_4)$ did not appear in the forward execution.

(B5) From $(C'_4,A'_4)$, process 3 is reversed to remove $(3,0)$, $\bot \xrightarrow{\mathtt{z}} (3,0)$, and $(1,0) \dashrightarrow^{\mathtt{x}} (3,0)$ to obtain $A_3$ and $[\mathtt{x},\mathtt{y},\mathtt{z} \mapsto 1,0,0]$.

(B6) Then, process 1 is reversed by removing $(1,0)$ and $\bot \xrightarrow{\mathtt{x}} (1,0)$ to obtain $A_2 = (\{\bot,(\varepsilon,0),(\varepsilon,1)\},\varnothing,\varnothing)$.

(B7) At last, process $\varepsilon$ reverses $b_2$ and $b_1$ to obtain $(C_{init},A_{init})$.

In (B4) step, there are two possibilities of reversing process 3 or process 2. In the above, $A_5$ is reversed by process 2 to $A'_4$ followed by process 3.

For a CRIL program $Pg$, let $B$ be the basic blocks in $Pg$. Let $O = \mathsf{PID} \times \bigcup_{b\in B} \mathsf{read}(b) \times \bigcup_{b\in B} \mathsf{write}(b)$. Proposition 2 ensures there is always a removable node along with removable edges.

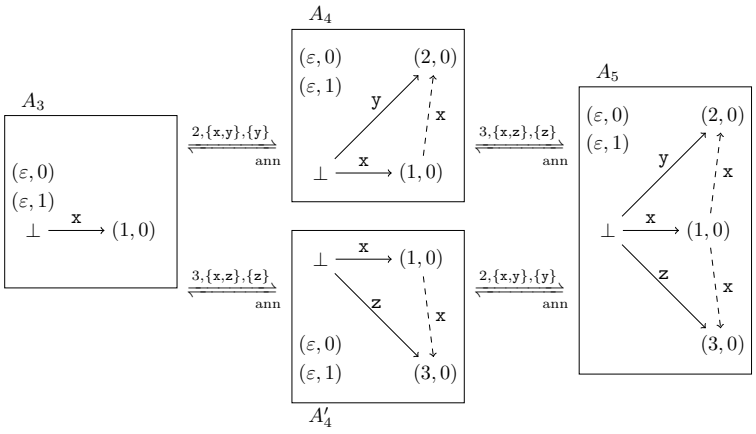



Figure 5: Annotation DAGs in backward execution

## 3.3 Properties for reversibility

We show that the operational semantics controlled by annotation DAG has proper properties for reversibility. We focus on the following two properties that are considered fundamental properties for reversibility [13].

**Causal Safety (CS):** An action can not be reversed until any actions caused by it have been reversed.

**Causal Liveness (CL):** We should allow actions to reverse in any order compatible with Causal Safety, not necessarily the exact inverse of the forward order.

[13] shows that those properties hold in an LTSI (LTS with Independence) provided that a small number of axioms are valid in the LTSI. We shall follow this approach by defining LTS from $\overset{p,Rd,Wt}{\rightleftharpoons}$ and add the independence relation to have the LTSI for the CRIL behavior. We will then show that the axioms for **CS** and **CL** hold.

**Definition 4.** $(\mathscr{C} \times \mathscr{A}, \mathsf{Lab}, \rightharpoonup)$ *is the* forward LTS for CRIL *where:*

- $\mathsf{Lab} = \mathsf{PID} \times 2^{\mathscr{R}} \times 2^{\mathscr{R}}$*; and*
- $(C,A) \xrightarrow{(p,Rd,Wt)} (C',A')$ *if* $(C,A) \overset{p,Rd,Wt}{\rightleftharpoons} (C',A')$

**Definition 5.** *The (combined) LTS for CRIL is* $(\mathscr{C} \times \mathscr{A}, \mathsf{Lab} \uplus \underline{\mathsf{Lab}}, \rightarrow)$ *where:*

- $\underline{\mathsf{Lab}} = \{\underline{(p,Rd,Wt)} \mid (p,Rd,Wt) \in \mathsf{Lab}\}$*; and*
- *For* $a \in \mathsf{Lab}$, $(C,A) \xrightarrow{a} (C',A')$ *iff* $(C,A) \xrightarrow{a} (C',A')$*, and* $(C,A) \xrightarrow{\underline{a}} (C',A')$ *iff* $(C',A') \xrightarrow{a} (C,A)$.

$\mathsf{Lab} \uplus \underline{\mathsf{Lab}}$ is ranged over by $\alpha, \beta, \cdots$, and $\mathsf{Lab}$ by $a, b, \cdots$. $\mathsf{und} : \mathsf{Lab} \uplus \underline{\mathsf{Lab}} \to \mathsf{Lab}$ where $\mathsf{und}(a) = a$ and $\mathsf{und}(\underline{a}) = a$. $\underline{\underline{a}} = a$. Given $t : P \xrightarrow{a} Q$, $\underline{t}$ is for $Q \xrightarrow{\underline{a}} P$.

For CRIL, the independence of transitions is defined as the independent memory update among concurrent processes. The processes running concurrently are not in the subprocess relation. Note that as pid $p \cdot 1$, $p \cdot 2$, $\cdots$ are assigned to the subprocesses of the process with pid of $p$. The process with the pid of $p$ is concurrent to the process with the pid of $q$ if $p \not\preceq q$ and $q \not\preceq p$. Hence, we give the dependence relation for labels as follows.

**Definition 6.** *For* $\alpha, \beta \in \mathsf{Lab} \uplus \underline{\mathsf{Lab}}$ *such that* $\mathsf{und}(\alpha) = (p_1, Rd_1, Wt_1)$ *and* $\mathsf{und}(\beta) = (p_2, Rd_2, Wt_2)$, $\alpha \iota_{\text{lab}} \beta$ *iff*

$$p_1 \not\preceq p_2 \wedge p_2 \not\preceq p_1 \wedge Rd_1 \cap Wt_2 = \varnothing \wedge Rd_2 \cap Wt_1 = \varnothing$$

The independence of transitions in LTS is defined as the transitions with independent labels. We define the Labeled Transition System with Independent transitions as the operational semantics of CRIL.

**Definition 7.** *For* $t : (C_1, A_1) \xrightarrow{\alpha} (C_1', A_1')$ *and* $u : (C_2, A_2) \xrightarrow{\beta} (C_2', A_2')$ *in the combined LTS for CRIL, t and u are* independent *of each other, written as* $t \iota u$ *if* $\alpha \iota_{\text{lab}} \beta$.

$(\mathscr{C} \times \mathscr{A}, \mathsf{Lab} \uplus \underline{\mathsf{Lab}}, \to, \iota)$ *is the* LTS of CRIL with independence.

In the sequel, we write '$LTSI_{CRIL}$' for the LTS of CRIL with independence.

### 3.3.1 Basic properties for reversibility

We take the axiomatic approach of [13], where the combination of the basic properties gives the proper reversibility. The first step is to show that the $LTSI_{CRIL}$ is *pre-reversible*. For this purpose, we show $LTSI_{CRIL}$ satisfies the following axioms: "**Square Property (SP)**", "**Backward Transitions are Independent (BTI)**", "**Well-Foundedness (WF)**", and "**Coinitial Propagation of Independence (CPI)**".

**Square Property(SP)** For $a \in \mathsf{Lab}$, when $C \xrightleftharpoons[\text{prog}]{a} C'$, we write $C \xrightarrow{a}_{\text{prog}} C'$ and $C' \xrightarrow{\underline{a}}_{\text{prog}} C$. Similarly, when $A \xrightleftharpoons[\text{ann}]{a} A'$, we write $A \xrightarrow{a}_{\text{ann}} A'$ and $A' \xrightarrow{\underline{a}}_{\text{ann}} A$.

By the definition of the independence transitions, the square property of the $\xrightarrow{\alpha}_{\text{prog}}$ is immediately shown.

**Proposition 3.** *Suppose* $C \xrightarrow{\alpha}_{\text{prog}} C'$, $C \xrightarrow{\beta}_{\text{prog}} C''$, *and* $\alpha \iota_{\text{lab}} \beta$. *Then there are the cofinal transitions* $C' \xrightarrow{\beta}_{\text{prog}} C'''$ *and* $C'' \xrightarrow{\alpha}_{\text{prog}} C'''$.

For annotation DAGs, we need to trace the difference of nodes and edges added or deleted by $\xrightarrow{\alpha}_{\text{ann}}$ to show the square property. We use the following notation to present differences in annotation DAGs:

$$\text{For } o : (V, E_R, E_W) \xrightarrow{\alpha}_{\text{ann}} (V', E_R', E_W'),\ \mathsf{diff}(o) = \begin{cases} (V' - V, E_R' - E_R, E_W' - E_W) & \text{if } \alpha \in \mathsf{Lab}, \\ (V - V', E_R - E_R', E_W - E_W') & \text{if } \alpha \in \underline{\mathsf{Lab}} \end{cases}$$

$$(V, E_R, E_W) \odot^{\alpha} (\Delta V, \Delta E_R, \Delta E_W) = \begin{cases} (V \cup \Delta V, E_R \cup \Delta E_R, E_W \cup \Delta E_W) & \text{if } \alpha \in \mathsf{Lab}, \\ (V - \Delta V, E_R - \Delta E_R, E_W - \Delta E_W) & \text{if } \alpha \in \underline{\mathsf{Lab}} \end{cases}$$

**Proposition 4.** *Let* $\mathsf{diff}(A \xrightarrow{\alpha}_{\text{ann}} A') = (\Delta V^{\alpha}, \Delta E_R^{\alpha}, \Delta E_W^{\alpha})$ *and* $\mathsf{diff}(A \xrightarrow{\beta}_{\text{ann}} A'') = (\Delta V^{\beta}, \Delta E_R^{\beta}, \Delta E_W^{\beta})$ *with* $\alpha \iota_{\text{lab}} \beta$. *Then,* $\Delta V^{\alpha} \cap \Delta V^{\beta} = \Delta E_R^{\alpha} \cap \Delta E_R^{\beta} = \Delta E_W^{\alpha} \cap \Delta E_W^{\beta} = \varnothing$.

*Proof.* For some $v_\alpha$ and $v_\beta$, $\Delta V^\alpha = \{v_\alpha\}$ and $\Delta V^\beta = \{v_\beta\}$. $\alpha \iota_{\text{lab}} \beta$ implies $v_\alpha \neq v_\beta$. All the edges of $\Delta E_R^\alpha \uplus \Delta E_W^\alpha$ come into $v_\alpha$ and all the edges of $\Delta E_R^\beta \uplus \Delta E_W^\beta$ come into $v_\beta$. Therefore, $\Delta V^\alpha \cap \Delta V^\beta = \Delta E_R^\alpha \cap \Delta E_R^\beta = \Delta E_W^\alpha \cap \Delta E_W^\beta = \varnothing$. □

**Proposition 5.** *Suppose $A \xrightarrow{a}_{\text{ann}} A'$ and $A \xrightarrow{\beta}_{\text{ann}} A''$ with $a \iota_{\text{lab}} \beta$. Then there is $A'''$ such that $A'' \xrightarrow{a}_{\text{ann}} A'''$ and $\mathsf{diff}(A \xrightarrow{a}_{\text{ann}} A') = \mathsf{diff}(A'' \xrightarrow{a}_{\text{ann}} A''')$.*

*Proof.* Assume $A = (V, E_R, E_W)$, $A'' = (V'', E_R'', E_W'')$, and $a = (p_a, Rd_a, Wt_a)$. $A'' \xrightarrow{a}_{\text{ann}} A'''$ for some $A'''$ since $a \in \mathsf{Lab}$. $a \iota_{\text{lab}} \beta$ implies that $\mathsf{max}_{p_a}(A) = \mathsf{max}_{p_a}(A'')$ and $\mathsf{last}(r, E_W) = \mathsf{last}(r, E_W'')$ for $r \in Rd_a$. Therefore, $\mathsf{diff}(A \xrightarrow{a}_{\text{ann}} A') = \mathsf{diff}(A'' \xrightarrow{a}_{\text{ann}} A''')$. □

**Proposition 6.** *Suppose $A \xrightarrow{\underline{a}}_{\text{ann}} A'$ and $A \xrightarrow{\beta}_{\text{ann}} A''$ with $\underline{a} \iota_{\text{lab}} \beta$. Then there is $A'''$ such that $A'' \xrightarrow{\underline{a}}_{\text{ann}} A'''$ and $\mathsf{diff}(A \xrightarrow{\underline{a}}_{\text{ann}} A') = \mathsf{diff}(A'' \xrightarrow{\underline{a}}_{\text{ann}} A''')$.*

*Proof.* Assume $\mathsf{diff}(A \xrightarrow{\beta}_{\text{ann}} A'') = (\Delta V^\beta, \Delta E_R^\beta, \Delta E_W^\beta)$, and $a = (p_a, Rd_a, Wt_a)$. Let $v = (p_a, \mathsf{max}_{p_a}(V))$. $\underline{a} \iota_{\text{lab}} \beta$ implies that no edges in $\Delta E_R^\beta \uplus \Delta E_W^\beta$ go out from $v$ and $v'$ such that $v' \overset{r}{\dashrightarrow} v$ in $A$. Therefore, $A'' \xrightarrow{\underline{a}}_{\text{ann}} A'''$ for some $A'''$. $\underline{a} \iota_{\text{lab}} \beta$ and $\underline{a} \in \underline{\mathsf{Lab}}$ derive $\mathsf{diff}(A \xrightarrow{\underline{a}}_{\text{ann}} A') = \mathsf{diff}(A'' \xrightarrow{\underline{a}}_{\text{ann}} A''')$. □

**Proposition 7.** *Suppose $A \xrightarrow{\alpha}_{\text{ann}} A'$ and $A \xrightarrow{\beta}_{\text{ann}} A''$ with $\alpha \iota_{\text{lab}} \beta$. Then $A'' \xrightarrow{\alpha}_{\text{ann}} A'''$, where $A''' = A'' \odot^\alpha \mathsf{diff}(A \xrightarrow{\alpha}_{\text{ann}} A')$.*

*Proof.* Proposition 5 and 6 derive $A'' \xrightarrow{\alpha}_{\text{ann}} A'''$. □

**Proposition 8.** *Suppose $A \xrightarrow{\alpha}_{\text{ann}} A'$, $A \xrightarrow{\beta}_{\text{ann}} A''$, and $\alpha \iota_{\text{lab}} \beta$. Then there are the cofinal transitions $A' \xrightarrow{\beta}_{\text{ann}} A'''$ and $A'' \xrightarrow{\alpha}_{\text{ann}} A'''$.*

*Proof.* By proposition 4, $\mathsf{diff}(A \xrightarrow{\alpha}_{\text{ann}} A')$ and $\mathsf{diff}(A \xrightarrow{\beta}_{\text{ann}} A'')$ are disjoint if $\alpha \iota_{\text{lab}} \beta$. Hence, the order of addition and deletion to/from $A$ does not affect the result. Therefore, $(A \odot^\alpha \mathsf{diff}(A \xrightarrow{\alpha}_{\text{ann}} A')) \odot^\beta \mathsf{diff}(A \xrightarrow{\beta}_{\text{ann}} A'') = (A \odot^\beta \mathsf{diff}(A \xrightarrow{\beta}_{\text{ann}} A'')) \odot^\alpha \mathsf{diff}(A \xrightarrow{\alpha}_{\text{ann}} A') = A'''$. By proposition 7, we have $A \xrightarrow{\alpha}_{\text{ann}} A' \xrightarrow{\beta}_{\text{ann}} A'''$ and $A \xrightarrow{\beta}_{\text{ann}} A'' \xrightarrow{\alpha}_{\text{ann}} A'''$ hold for such $A'''$. □

Combining proposition 3 with proposition 8 by **ProgAnn**, the square property holds.

**Lemma 1** (Square Property)**.** *Whenever $t : (C_P, A_P) \xrightarrow{\alpha} (C_Q, A_Q)$, $u : (C_P, A_P) \xrightarrow{\beta} (C_R, A_R)$, and $t \iota u$, then there are cofinal transitions $u' : (C_Q, A_Q) \xrightarrow{\beta} (C_S, A_S)$, and $t' : (C_R, A_R) \xrightarrow{\alpha} (C_S, A_S)$.*

**Backward Transitions are Independent (BTI)** **BTI** is useful for reversibility because an available backward transition does not depend on any other backward transition. In CRIL, a label of $LTSI_{CRIL}$ gives the information to establish BTI.

**Lemma 2** (Backward Transitions are Independent)**.** *Whenever $t : (C_P, A_P) \xrightarrow{a} (C_Q, A_Q)$, $u : (C_P, A_P) \xrightarrow{b} (C_R, A_R)$, and $t \neq u$, then $t \iota u$.*

*Proof.* Assume $A_P = (V, E_R, E_W)$, $a = (p_a, Rd_a, Wt_a)$, and $b = (p_b, Rd_b, Wt_b)$. Let $v_a = (p_a, \mathsf{max}_{p_a}(V))$ and $v_b = (p_b, \mathsf{max}_{p_b}(V))$.

Assume $p_a \preceq p_b$. Then $p_a = p_b$ holds from the operational semantics. $p_a = p_b$ derives $t = u$, which contradicts $t \neq u$. Therefore, $p_a \not\preceq p_b$ holds. Similarly, $p_b \not\preceq p_a$ also holds.

Assume $Rd_a \cap Wt_b \neq \varnothing$. There exists $r \in Rd_a \cap Wt_b$. If $r \in Wt_a$, then $\mathsf{last}(r, E_W) = v_a$ and $\mathsf{last}(r, E_W) = v_b$. Therefore $p_a = p_b$, however it contradicts $p_a \not\preceq p_b$. If $r \notin Wt_a$, then $\mathsf{last}(r, E_W) \overset{r}{\dashrightarrow} v_a \in E_R$. $r \in Wt_b$ derives $\mathsf{last}(r, E_W) = v_b$. Therefore $v_b \overset{r}{\dashrightarrow} v_a \in E_R$, however it contradicts that no edges go out from $v_b$ derived from $u$. Therefore $Rd_a \cap Wt_b = \varnothing$. Similarly, $Rd_b \cap Wt_a = \varnothing$ also holds. □

**Well-Foundedness (WF)** For a backward transition $(C, A) \xrightarrow{a} (C', A')$, the number of nodes of $A'$ is strictly less than that of $A$. Since the number of vertices of annotation DAG is finite, it is not possible to remove vertices infinitely.

**Coinitial Propagation of Independence (CPI)** Given a commuting square with independence at one corner, CPI allows us to deduce independence between coinitial transitions at the other three corners.

**Lemma 3** (Coinitial Propagation of Independence)**.** *Suppose* $t : (C_P, A_P) \xrightarrow{\alpha} (C_Q, A_Q)$, $u : (C_P, A_P) \xrightarrow{\beta} (C_R, A_R)$, $u' : (C_Q, A_Q) \xrightarrow{\beta} (C_S, A_S)$, $t' : (C_R, A_R) \xrightarrow{\alpha} (C_S, A_S)$, *and* $t \iota u$. *Then* $u' \iota \underline{t}$.

*Proof.* $t \iota u$ implies $\alpha \iota_{\mathrm{lab}} \beta$. Since $\beta \iota_{\mathrm{lab}} \underline{\alpha}$, $u' \iota \underline{t}$. □

### 3.3.2 Events

The properties above make $LTSI_{CRIL}$ pre-reversible. Next, we check if $LTSI_{CRIL}$ can derive events for establishing reversibility. Following [13], events in $LTSI_{CRIL}$ are derived as an equivalence over transitions.

**Definition 8.** *Let* $\sim$ *be the smallest equivalence relation on transitions satisfying: if* $t : (C_P, A_P) \xrightarrow{\alpha} (C_Q, A_Q)$, $u : (C_P, A_P) \xrightarrow{\beta} (C_R, A_R)$, $u' : (C_Q, A_Q) \xrightarrow{\beta} (C_S, A_S)$, $t' : (C_R, A_R) \xrightarrow{\alpha} (C_S, A_S)$, *and* $t \iota u$, *then* $t \sim t'$. *The equivalence classes of forward transitions* $[(C_P, A_P) \xrightarrow{a} (C_Q, A_Q)]$, *are the* events. *The equivalence classes of backward transitions* $[(C_P, A_P) \xrightarrow{\underline{a}} (C_Q, A_Q)]$, *are the* reverse events.

Given $\gamma = \alpha_1 \cdots \alpha_n \in (\mathsf{Lab} \uplus \underline{\mathsf{Lab}})^*$, a sequence of transitions $(C_0, A_0) \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} (C_n, A_n)$ is written as $s : (C_0, A_0) \xrightarrow{\gamma}_* (C_n, A_n)$.

Since the transitions of program configurations in $LTSI_{CRIL} \xrightarrow{\alpha}_{\mathrm{prog}}$ have no control for reversibility, events are substantially derived from the operations of annotation DAGs.

**Definition 9.** *Let* $\sim_{\mathrm{ann}}$ *be the smallest equivalence relation over operations of annotation DAGs satisfying: if* $o_1 : A_P \xrightarrow{\alpha}_{\mathrm{ann}} A_Q$, $o_2 : A_P \xrightarrow{\beta}_{\mathrm{ann}} A_R$, $o_2' : A_Q \xrightarrow{\beta}_{\mathrm{ann}} A_S$, $o_1' : A_R \xrightarrow{\alpha}_{\mathrm{ann}} A_S$, *and* $\alpha \iota_{\mathrm{lab}} \beta$, *then* $o_1 \sim o_1'$. $[A \xrightarrow{a}_{\mathrm{ann}} A']_{\mathrm{ann}}$ *and* $[A \xrightarrow{\underline{a}}_{\mathrm{ann}} A']_{\mathrm{ann}}$ *are the forward and backward equivalence classes by* $\sim_{\mathrm{ann}}$.

**Proposition 9.** *For* $t : (C_P, A_P) \xrightarrow{\alpha} (C_Q, A_Q)$ *and* $t' : (C_R, A_R) \xrightarrow{\alpha} (C_S, A_S)$, *the following holds.*
$t \sim t'$ *iff* $o \sim_{\mathrm{ann}} o'$ *and* $\exists \gamma.(C_P, A_P) \xrightarrow{\gamma}_* (C_R, A_R)$ *where* $o : A_P \xrightarrow{\alpha}_{\mathrm{ann}} A_Q$ *and* $o' : A_R \xrightarrow{\alpha}_{\mathrm{ann}} A_S$.

Intuitively, operations for annotation DAGs are independent if they add or remove nodes and edges at unrelated places. If $o_1 \sim_{\mathrm{ann}} o_2$, then $o_1$ and $o_2$ add or remove the same fragment of annotation DAGs to or from the nodes of the same causality. In $LTSI_{CRIL}$, the equivalence over operations of annotation DAGs is considered as an *event*. This shows that events for reversibility are consistently defined over $LTSI_{CRIL}$, meaning the operational semantics is detailed enough to give the **IRE** property as follows, which is necessary for our objectives.

**Independence Respects Events (IRE)**

**Lemma 4** (Independence Respects Events)**.** *Suppose $t \sim t' \iota u$. Then $t \iota u$.*

*Proof.* If $t \sim t'$, $t$ has the same label as $t'$. Then, $t \iota u$. □

### 3.3.3 Causal Safety and Causal Liveness

Let $\sharp(s, [A \xrightarrow{a} A']_{\text{ann}})$ be the number of occurrences of transitions $t$ in $s$ such that $t \in [(C,A) \xrightarrow{a} (C',A')]$, minus the number of occurrences of transitions $t$ in $s$ such that $t \in [(C,A) \xrightarrow{a} (C',A')]$.

Using the result of [13], the properties of **SP**(Lemma 1), **BTI**(Lemma 2), **WF**, **CPI**(Lemma 3), and **IRE** (Lemma 4) make **Causal Safety (CS)** and **Causal Liveness (CL)** hold. Due to the fact that the causality is stored in the annotation DAGs, the properties can be stated in $LTSI_{CRIL}$ as below.

**Theorem 1** (Causal Safety)**.** *Whenever $(C_P,A_P) \xrightarrow{a} (C_Q,A_Q)$, $s : (C_Q,A_Q) \xrightarrow{\gamma}_* (C_R,A_R)$ with $\sharp(s, [A_P \xrightarrow{a} A_Q]_{\text{ann}}) = 0$, and $(C_S,A_S) \xrightarrow{a} (C_R,A_R)$ then $(C_P,A_P) \xrightarrow{a} (C_Q,A_Q) \iota t$ for all $t$ in $s$ such that $\sharp(s, [A_P \xrightarrow{a} A_Q]_{\text{ann}}) > 0$.*

**Theorem 2** (Causal Liveness)**.** *Whenever $(C_P,A_P) \xrightarrow{a} (C_Q,A_Q)$, $s : (C_Q,A_Q) \xrightarrow{\gamma}_* (C_R,A_R)$, $\sharp(s, [A_P \xrightarrow{a} A_Q]) = 0$, and $(C_P,A_P) \xrightarrow{a} (C_Q,A_Q) \iota t : (C,A) \xrightarrow{b} (C',A')$ for all $t$ in $s$ such that $\sharp(s, [A \xrightarrow{a} A']) > 0$ with $(C_P,A_P) \xrightarrow{a} (C_Q,A_Q) \sim (C_S,A_S) \xrightarrow{a} (C_R,A_R)$, then we have $(C_S,A_S) \xrightarrow{a} (C_R,A_R)$ with $(C_P,A_P) \xrightarrow{a} (C_Q,A_Q) \sim (C_S,A_S) \xrightarrow{a} (C_R,A_R)$.*

Based on these properties, $LTSI_{CRIL}$ can be implemented correctly with the pointers for processes managed by a process map along with annotation DAGs as the operational semantics of CRIL.

# 4 Example: Airline ticketing

We show a version of the airline ticketing program [5] in CRIL in figure 6. Two agents attempt to sell three seats of an airline. This program has a data race for variable `seats` of the remaining seats because two agents may check the remaining seats simultaneously before making sales. Since the data race does not always happen, it is useful to roll back to the point where checking remaining seats is insufficient. Here, `agent1` and `agent2` are used to record the number of tickets sold by each agent.

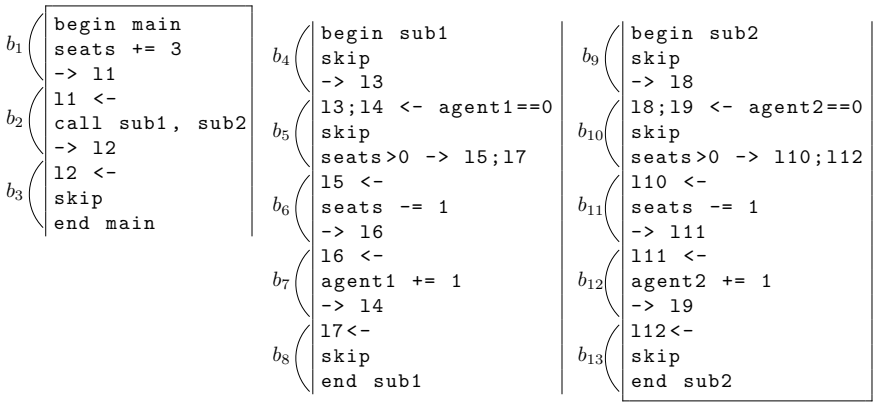



Figure 6: An airline ticketing program in CRIL

| basic block | | seats | agent1 | agent2 |
|---|---|---|---|---|
| $(\varepsilon,0)$ | $b_1$ | 3 | 0 | 0 |
| $(\varepsilon,1)$ | $b_2$ | 3 | 0 | 0 |
| $(1,0)$ | $b_4$ | 3 | 0 | 0 |
| $(2,0)$ | $b_9$ | 3 | 0 | 0 |
| $(1,1)$ | $b_5$ | 3 | 0 | 0 |
| $(1,2)$ | $b_6$ | 2 | 0 | 0 |
| $(1,3)$ | $b_7$ | 2 | 1 | 0 |
| $(2,1)$ | $b_{10}$ | 2 | 1 | 0 |
| $(2,2)$ | $\mathbf{b_{11}}$ | 1 | 1 | 0 |
| $(2,3)$ | $b_{12}$ | 1 | 1 | 1 |
| $(2,4)$ | $\mathbf{b_{10}}$ | 1 | 1 | 1 |
| $(1,4)$ | $\mathbf{b_5}$ | 1 | 1 | 1 |
| $(2,5)$ | $b_{11}$ | 0 | 1 | 1 |
| $(1,5)$ | $b_6$ | -1 | 1 | 1 |
| $(2,6)$ | $b_{12}$ | -1 | 1 | 2 |
| $(2,7)$ | $b_{10}$ | -1 | 1 | 2 |
| $(2,8)$ | $b_{13}$ | -1 | 1 | 2 |
| $(1,6)$ | $b_7$ | -1 | 2 | 2 |
| $(1,7)$ | $b_5$ | -1 | 2 | 2 |
| $(1,8)$ | $b_8$ | -1 | 2 | 2 |
| $(\varepsilon,2)$ | $b_2$ | -1 | 2 | 2 |
| $(\varepsilon,3)$ | $b_3$ | -1 | 2 | 2 |

Table 3: A faulty execution

Table 3 shows a forward execution that ends `seats` $= -1$. Figure 7 is the annotation DAG when terminated at `'end main'` in $b_3$. To investigate the cause of the data race, we focus on the edges labeled with `seats`. The solid edges indicate that `seats` is written in $(\varepsilon,0)$, $(1,2)$, $(2,2)$, $(2,5)$, and $(1,5)$. In particular, `seats` defined at $(2,2)$ is used to update by processes 2 and 3 to cause the data race. (The steps in bold are involved in the problem.) To resolve the data race, each value of `seats` should be checked exactly once, except for the last value of `seats`.

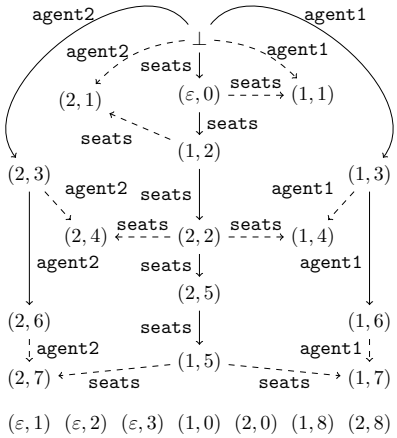



Figure 7: The annotation DAG after the forward execution with the data race

Figure 8 shows the airline program where `sub1` and `sub2` are replaced by those with the V-P operations. The parameter of the V-P operations works as a semaphore to check and update `seats` as a critical region. Figure 9 is the annotation DAG by the forward execution with `sub1` done first once and then `sub2` done twice. Process 1 executes $b'_5$ setting `semaphore` $= 1$ at $(1,1)$ first. (`sem` is for `semaphore` in the figure.) This prevents process 2 executing $b'_{10}$ at $(2,1)$ since `semaphore` must be 0. Backwards, $b'_{14}$ and $b'_{15}$ work as `V semaphore`. In the backward execution, the order of basic blocks is stored in the annotation DAG. It works as follows:

- The sequence of $\xrightarrow{\texttt{sem}}$ is alternatively from `V` and `P` operations in the forward execution. $\bot \xrightarrow{\texttt{sem}} (1,1)$ is by $b'_5$ and $(1,1) \xrightarrow{\texttt{sem}} (1,3)$ by $b'_{14}, \cdots, (1,3) \xrightarrow{\texttt{sem}} (2,1)$ by $b'_{10}$, $(2,1) \xrightarrow{\texttt{sem}} (2,3)$ by $b'_{15}, \cdots$.
- When `seats` $= 0$, `semaphore` is released with no operation. $(2,7) \xrightarrow{\texttt{sem}} (1,5) \xrightarrow{\texttt{sem}} (1,6)$ by $b'_5$ and $b'_8$ and $(1,6) \xrightarrow{\texttt{sem}} (2,9) \xrightarrow{\texttt{sem}} (2,10)$ by $b'_{10}$ and $b'_{13}$.
- In backward, `sub2` is ready since $(2,10)$ is $\mathsf{last}(E_W, \texttt{sem})$.
- Then, `sub1` is done with no operation and $(2,7)$ is `P` in `sub2`. The order of `V` and `P` is kept until reaching $\bot$.

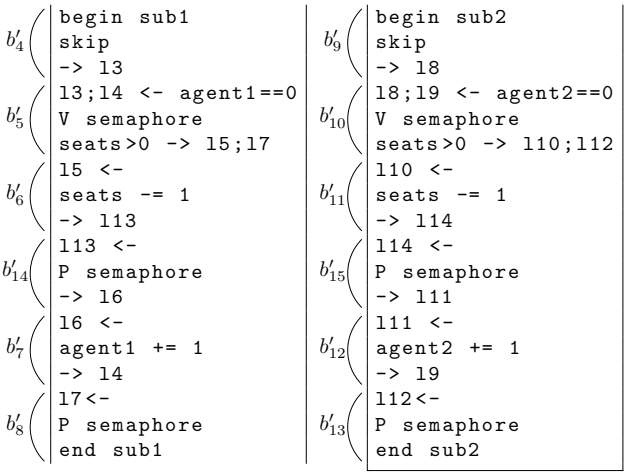



Figure 8: An airline ticketing with semaphore

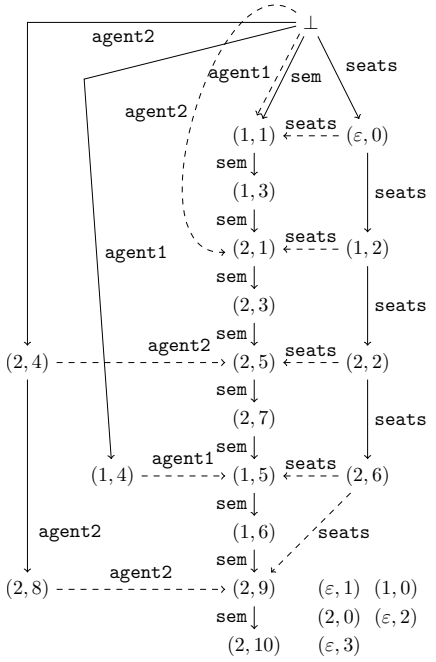



Figure 9: The annotation DAG after the forward execution with semaphore

# 5 Concluding remarks

We have proposed CRIL as a reversible concurrent intermediate language. CRIL is an extension of RIL [16] to enable running multiple subroutines as processes running in parallel. CRIL is intended to be fairly low-level in that each instruction is at a level similar to the three-address codes to mediate the translation from a high-level program to a machine-oriented code. The operational semantics of CRIL defined as $LTSI_{CRIL}$ is shown to have the properties of Causal Safety and Causal Liveness under the independence of concurrent processes and shared memory update. By the result of [13], $LTSI_{CRIL}$ also satisfies other properties: Parabolic lemma, Causal Consistency, Unique Transition, and Independence of Diamonds.

As related work, [1] provides a compiler from ROOPL++ to PISA [22] with no intermediate language, where the translation from an object-oriented source program to the low-level PISA code is a big task. [6] proposes an annotation to a concurrent imperative program while executing forward, where the annotation is attached directly to the source program for reversing the execution. [7] investigates its properties of reversibility. CRIL uses a similar idea as Hoey's, but CRIL is at a rather lower level to provide a finer granularity for detailed analysis in translation, such as optimization. [8] presents a collection of simple stack machines with a fork and merge mechanism, where the causality is embedded in the runtime.

For future work, we have focused only on the fundamental properties. We will investigate further how more properties in reversibility contribute to behavioral analysis for concurrent programs. Currently, the dependency of the heap memory `M` is treated as one memory resource. More detailed dependency is necessary for practical use. Deriving the optimization technique in the front-end part of compilers is future work via the reversible version of SSA, such as RSSA [17] for concurrent imperative programs. CRIL is based on the shared memory model. Incorporating channel-based communications is also future work to use for the message-passing model like Erlang [12].

**Acknowledgement** We thank Dr. Irek Ulidowski of the University of Leicester for giving valuable suggestions to the draft. We also thank Prof. Nobuko Yoshida of the University of Oxford, Prof. Hiroyuki Seki, Prof. Koji Nakazawa, and Prof. Yuichi Kaji of Nagoya University for fruitful discussions. We thank the anonymous reviewers for providing fruitful comments. This work is supported by JSPS Kakenhi 21H03415.

# References

[1] Martin Holm Cservenka, Robert Glück, Tue Haulund & Torben Ægidius Mogensen (2018): *Data Structures and Dynamic Memory Management in Reversible Languages*. In: *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings*, *Lecture Notes in Computer Science* 11106, Springer, pp. 269–285, doi:10.1007/978-3-319-99498-7_19.

[2] Vincent Danos & Jean Krivine (2004): *Reversible Communicating Systems*. In: *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, *Lecture Notes in Computer Science* 3170, Springer, pp. 292–307, doi:10.1007/978-3-540-28644-8_19.

[3] Niklas Deworetzki, Martin Kutrib, Uwe Meyer & Pia-Doreen Ritzke (2022): *Optimizing Reversible Programs*. In: *RC 2022*, *Lecture Notes in Computer Science* 13354, Springer, pp. 224–238, doi:10.1007/978-3-031-09005-9_16.

[4] Lasse Hay-Schmidt, Robert Glück, Martin Holm Cservenka & Tue Haulund (2021): *Towards a Unified Language Architecture for Reversible Object-Oriented Programming*. In: *Reversible Computation - 13th*

*International Conference, RC 2021, Virtual Event, July 7-8, 2021, Proceedings*, *Lecture Notes in Computer Science* 12805, Springer, pp. 96–106, doi:10.1007/978-3-030-79837-6_6.

[5] James Hoey, Ivan Lanese, Naoki Nishida, Irek Ulidowski & Germán Vidal (2020): *A Case Study for Reversible Computing: Reversible Debugging of Concurrent Programs*. In: *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*, *Lecture Notes in Computer Science* 12070, Springer, pp. 108–127, doi:10.1007/978-3-030-47361-7_5.

[6] James Hoey & Irek Ulidowski (2022): *Reversing an imperative concurrent programming language*. *Sci. Comput. Program.* 223, p. 102873, doi:10.1016/j.scico.2022.102873.

[7] James Hoey & Irek Ulidowski (2022): *Towards Causal-Consistent Reversibility of Imperative Concurrent Programs*. In: *Reversible Computation - 14th International Conference, RC 2022, Urbino, Italy, July 5-6, 2022, Proceedings*, *Lecture Notes in Computer Science* 13354, Springer, pp. 213–223, doi:10.1007/978-3-031-09005-9_15.

[8] Takashi Ikeda & Shoji Yuen (2020): *A Reversible Runtime Environment for Parallel Programs*. In: *RC 2020*, *Lecture Notes in Computer Science* 12227, Springer, pp. 272–279, doi:10.1007/978-3-030-52482-1_18.

[9] Martin Kutrib, Uwe Meyer, Niklas Deworetzki & Marc Schuster (2021): *Compiling Janus to RSSA*. In: *Reversible Computation - 13th International Conference, RC 2021, Virtual Event, July 7-8, 2021, Proceedings*, *Lecture Notes in Computer Science* 12805, Springer, pp. 64–78, doi:10.1007/978-3-030-79837-6_4.

[10] Ivan Lanese, Doriana Medic & Claudio Antares Mezzina (2021): *Static versus dynamic reversibility in CCS*. *Acta Informatica* 58(1-2), pp. 1–34, doi:10.1007/s00236-019-00346-6.

[11] Ivan Lanese, Claudio Antares Mezzina & Jean-Bernard Stefani (2010): *Reversing Higher-Order Pi*. In: *CONCUR 2010*, *Lecture Notes in Computer Science* 6269, Springer, pp. 478–493, doi:10.1007/978-3-642-15375-4_33.

[12] Ivan Lanese, Naoki Nishida, Adrián Palacios & Germán Vidal (2018): *A theory of reversibility for Erlang*. *J. Log. Algebraic Methods Program.* 100, pp. 71–97, doi:10.1016/j.jlamp.2018.06.004.

[13] Ivan Lanese, Iain C. C. Phillips & Irek Ulidowski (2020): *An Axiomatic Approach to Reversible Computation*. In: *Proceedings of FOSSACS 2020*, *Lecture Notes in Computer Science* 12077, Springer, pp. 442–461, doi:10.1007/978-3-030-45231-5_23. (The full version is at `https://arxiv.org/abs/2307.13360`.).

[14] Chris Lattner & Vikram S. Adve (2004): *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. In: *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, IEEE Computer Society, pp. 75–88, doi:10.1109/CGO.2004.1281665.

[15] Hernán C. Melgratti, Claudio Antares Mezzina & G. Michele Pinna (2021): *A distributed operational view of Reversible Prime Event Structures*. In: *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021, Rome, Italy, June 29 - July 2, 2021*, IEEE, pp. 1–13, doi:10.1109/LICS52264.2021.9470623.

[16] Torben Ægidius Mogensen (2015): *Garbage Collection for Reversible Functional Languages*. In: *Reversible Computation - 7th International Conference, RC 2015, Grenoble, France, July 16-17, 2015, Proceedings*, *Lecture Notes in Computer Science* 9138, Springer, pp. 79–94, doi:10.1007/978-3-319-20860-2_5.

[17] Torben Ægidius Mogensen (2015): *RSSA: A Reversible SSA Form*. In: *Perspectives of System Informatics - 10th International Andrei Ershov Informatics Conference, PSI 2015*, *Lecture Notes in Computer Science* 9609, Springer, pp. 203–217, doi:10.1007/978-3-319-41579-6_16.

[18] Iain Phillips & Irek Ulidowski (2014): *Event Identifier Logic*. *Math. Struct. Comput. Sci.* 24(2), doi:10.1017/S0960129513000510.

[19] Iain Phillips & Irek Ulidowski (2015): *Reversibility and asymmetric conflict in event structures*. *J. Log. Algebraic Methods Program.* 84(6), pp. 781–805, doi:10.1016/j.jlamp.2015.07.004.

[20] Iain C. C. Phillips & Irek Ulidowski (2007): *Reversing algebraic process calculi*. *J. Log. Algebraic Methods Program.* 73(1-2), pp. 70–96, doi:10.1016/j.jlap.2006.11.002.

[21] Irek Ulidowski, Iain Phillips & Shoji Yuen (2018): *Reversing Event Structures*. *New Gener. Comput.* 36(3), pp. 281–306, doi:10.1007/s00354-018-0040-8.

[22] Carlin Vieri (1999): *Reversible computer engineering and architecture*. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA. Available at `https://hdl.handle.net/1721.1/80144`.

[23] Tetsuo Yokoyama (2010): *Reversible Computation and Reversible Programming Languages*. *Electron. Notes Theor. Comput. Sci.* 253(6), pp. 71–81, doi:10.1016/j.entcs.2010.02.007.

[24] Tetsuo Yokoyama, Holger Bock Axelsen & Robert Glück (2011): *Towards a Reversible Functional Language*. In: *RC 2011*, *Lecture Notes in Computer Science* 7165, Springer, pp. 14–29, doi:10.1007/978-3-642-29517-1_2.