EPTCS 425

Proceedings of the

18th

Interaction and Concurrency Experience

Lille, France, 20th June 2025

Edited by: Clément Aubert, Cinzia Di Giusto, Simon Fowler and Violet Ka I Pun

Published: 19th August 2025
DOI: 10.4204/EPTCS.425
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

Clément Aubert
Augusta University, USA

Cinzia Di Giusto
Université Côte d'Azur, CNRS, I3S Sophia Antipolis, FR

Simon Fowler
University of Glasgow School of Computing Science, GB

Violet Ka I Pun
Western Norway University of Applied Sciences, NO

This volume contains the proceedings of ICE'25, the 18th Interaction and Concurrency Experience, which was held in Lille, France, as a satellite event of DisCoTec'25. The previous editions of ICE are now conveniently listed on the series website, which also links to ICE's dblp page. The conference ICE is co-located with since 2010, DisCoTec, also continued its efforts to collect and link to previous editions, sometimes using the services of the Wayback Machine.

The ICE workshop series features a distinguishing review and selection procedure: PC members are encouraged to interact with authors in a double anonymous environment. This year again, these interactions took place on the HotCRP platform which combines paper selection features with forum-like interactions. As in the past editions, the forum discussion during the review and selection phase of ICE'25 considerably improved the accuracy of the feedback from the reviewers and the quality of accepted papers[1] and offered the basis for lively discussion during the workshop. The time and effort spent on the interaction between reviewers and authors is rewarding for all parties. The discussions on HotCRP made it possible to resolve misunderstandings at an early stage of the review process, to discover and correct mistakes in key definitions, and to improve examples and presentation.

ICE was glad to receive 7 submissions, including one that was withdrawn and a short presentation that is not part of this proceedings. Each paper was reviewed by three PC members and about 75 comments were made in total, witnessing very lively discussions between PC members and authors, and during the discussion phase. We were proud to host one invited talk by Kirstin Peters. The abstract of her invited talk is included in this volume, together with the final versions of the research papers, which take into account the discussion at the workshop and after.

We would like to thank the 14 authors of all the submitted papers for their interest in the workshop. We thank Kirstin Peters for accepting our invitations to present her work. We are extremely grateful for the efforts of the PC members:

- Franco Barbanera (Dept. of Mathematics and Computer Science - University of Catania)
- Manel Barkallah (University of Namur)
- Matteo Cimini (University of Massachusetts Lowell)
- Farzaneh Derakhshan (Illinois Tech)
- Emanuele D'Osualdo (University of Konstanz)
- **Luc Edixhoven (University of Southern Denmark)** ⋆
- Lorenzo Gheri (University of Liverpool)
- Lucie Guillou (IRIF, Université Paris Cité)
- Ping Hou (University of Oxford)

[1] As well, we hope, as the quality of future resubmission in the case of papers being rejected.

C. Aubert, C. Di Giusto, S. Fowler & V. K. I Pun (Eds.):
18th Interaction and Concurrency Experience (ICE 2025)
EPTCS 425, 2025, pp. ii–iii, doi:10.4204/EPTCS.425.0

- Andrew K. Hirsch (University at Buffalo, SUNY)
- Jonas Kastberg Hinrichsen (IT University of Copenhagen)
- Matthew Alan Le Brun (University of Glasgow)
- Andreia Mordido (LASIGE, University of Lisbon)
- Maurizio Murgia (Gran Sasso Science Institute)
- Jonah Pears (University of Kent)
- Felix Stutz (University of Luxembourg)
- Petra van den Bos (Formal Methods and Tools group (FMT) (University of Twente)
- Bas van den Heuvel (HKA Karlsruhe and University of Freiburg)

⋆ **The *ICE 2025 Outstanding PC Member Award* was awarded this year to Luc Edixhoven!**

Previously, were awarded:

- ICE 2024 Outstanding PC Member Award: *Bas van den Heuvel*
- ICE 2023 Outstanding PC Member Award: *Sergueï Lenglet*
- ICE 2022 Outstanding PC Member Award: *Duncan Paul Attard*
- ICE 2021 Outstanding PC Member Award: *Ivan Prokić*

We thank the DisCoTec'25 organizers, in particular the General Chair of the Organizing Committee, Simon Bliudze, for providing a welcoming, inclusive and lively environment for the preparation and staging of the event. We thank the editors of EPTCS for the publication of these post-proceedings. Cinzia and Clément are particularly thankful for the trust put in them by the ICE steering committee during their 4-years mandate, and wish Simon and Violet all the best in the organization of future editions–it has been a pleasure working with you all.

On the Expressiveness of MPST

Kirstin Peters
University of Augsburg, Germany
kirstin.peters@uni-a.de

Multiparty session types (MPST) are a type discipline for enforcing the structured, deadlock-free communication of concurrent and message-passing programs. In this talk we will analyse the expressive power of MPST. In particular, we are interested in features that mark the difference expressive power of synchronous and asynchronous distributed languages. In the synchronous pi-calculus mixed choice is the main ingredient for its expressive power. Traditional MPST have in contrast usually a limited form of choice, in which alternative communication possibilities are offered by a single participant and selected by another. Accordingly, we extend MPST by a more general mixed choice construct.

Biography Kirstin Peters studied computer science at the university of Potsdam, did her PhD at the technical university of Berlin. After some stops at Uppsala University and the technical university of Dresden, she had a junior professorship at the technical university of Darmstadt. Since April 2022, she is a professor at the university of Augsburg.

C. Aubert, C. Di Giusto, S. Fowler & V. K. I Pun (Eds.):
18th Interaction and Concurrency Experience (ICE 2025)
EPTCS 425, 2025, pp. 1–1, doi:10.4204/EPTCS.425.1

Modular Multiparty Sessions with Mixed Choice

Franco Barbanera *
Dipartimento di Matematica e Informatica, Università di Catania, Catania, Italy
franco.barbanera@unict.it

Mariangiola Dezani-Ciancaglini
Dipartimento di Informatica, Università di Torino, Torino, Italy
dezani@di.unito.it

MultiParty Session Types (MPST) provide a useful framework for safe concurrent systems. *Mixed choice* (enabling a participant to play at the same time the roles of sender and receiver) increases the expressive power of MPST as well as the difficulty in controlling safety of communications. Such a control is more viable when modular systems are considered and the power of mixed choice fully exploited only inside loosely coupled modules. We carry over such idea in a type assignment approach to multiparty sessions. Typability for modular sessions entails Subject Reductions, Session Fidelity and Lock Freedom.

Keywords: Multiparty Sessions, Modular Systems, Global Types.

1 Introduction

MultiParty Session Types (MPST) offer a structured approach to the development and formal verification of concurrent and distributed systems [18, 19]. As in the vast majority of choreographic formalisms, two distinct but related views of concurrent systems are taken into account: (a) the *global view*, a formal specification via *global types* of the overall behaviour of a system; (b) the *local view*, namely a description, at different levels of abstraction, of the behaviours of the single components. A key issue in MPST, and choreographies in general, is the relation between these two views. Among others, we can refer to the notion of *projection*, used till recently in most of the MPST formalisms. Given a (well-formed) global type, the projection operator produces a tuple of local types – one for each component – which generalises binary session types [16, 17]. Such local types can be looked at as an abstraction of finer grained descriptions of processes. Another approach to the MPST global-local relationship is the one embodied in the so called *Simple MultiParty Sessions* (SMPS) formalisms. Such an approach was first introduced in [12] and [4], and further investigated in a bunch of papers, among which [3, 7, 5, 6, 1, 2]. Whereas the MPST approach typically considers two-layered local views – a layer of processes and a layer of local types – SMPS are based on single-layered local views, where only a fairly abstract notion of process is considered. In SMPS, which is the general setting of the present paper, systems of communicating processes are represented as *multiparty sessions*, i.e. parallel compositions of named processes (the *participants*). Then, by means of type systems, global types are inferred for such sessions. Typability is such to ensure relevant communication properties – typically Lock Freedom – for sessions. Besides the above mentioned ones, it is worth recalling that also other approaches have been investigated, like the one introduced in [32], where the global view is only implicitly considered.

*Partially supported by Project "National Center for HPC, Big Data e Quantum Computing", Programma M4C2, Investimento 1.3 – Next Generation EU; and by the PIAno di inCEntivi per la RIcerca di Ateneo 2024-2026 UniCT (Linea di Intervento 1).

C. Aubert, C. Di Giusto, S. Fowler & V. K. I Pun (Eds.):
18th Interaction and Concurrency Experience (ICE 2025)
EPTCS 425, 2025, pp. 2–20, doi:10.4204/EPTCS.425.2

A common feature of all the above mentioned approaches, till lately, has been the use of communication models where, before any interaction, a process can clearly be identified as a sender or a receiver. The intrinsic potentiality of nondeterministically choosing among both inputs and outputs inside a single process interaction (usually referred to in the literature as "mixed choice") has however recently intrigued session type researchers, both for the binary and the multiparty cases [10, 28, 29, 30, 31]. A thorough investigation of the expressivity of mixed choice in (synchronous) MPST formalisms has been carried on in [31]. For instance, mixed choice enables to implement protocols safely exploiting circular interactions, as shown in [31] through an example recalled in the present paper (see Example 2.4). This is not possible in usual MPST formalisms where typing [7] or, equivalently, the projectability condition on global types – as shown in [6] – consists essentially in checking the possibility of sequentialising the interactions in a protocol. Together with its expressive power, however, mixed choice brings subtly harmful features, as already exposed decades ago [15] in the setting of Communicating Finite State Machines [9] (an asynchronous formalism closely related to MPST). In the present paper we aim at exploiting such expressive power in a safe and controlled way using a SMPS setting. In order to do that we resort to the notion of *modularity*.

Modularity is a fairly general property of complex systems. Any complex system can be decomposed into smaller subsystems that are always going to be interdependent to some extent and independent to some other extent [33]. In fact, in many human activities, from business to biology, as well as to software engineering, modularisation offers a strategic approach enabling to cope with their complexity. Modularity in software engineering refers to the design approach that emphasises the separation of concerns: a complex software system is decomposed into smaller, loosely coupled modules, where coupling is the degree of interdependence between the modules. By means of project modularisation one manages to, among others, reduce complexity (breaking down a large system into smaller modules makes it more manageable and easier to understand [25]) as well as to improve testing and separation of concerns (SoC), a fundamental principle in software engineering. In particular, in modular programming, concerns are separated such that modules, performing logically coherent tasks, do interact through simple and manageable interfaces.

Our proposal is hence to restrict our attention to sessions corresponding to modularised systems. A type discipline is then proposed that profits, as in more rigid MPST formalisms, from a form of "sequentialisation" condition. Such a condition however, instead of being imposed on participants, is imposed on the modules forming a session, inside which the mixed choice can be freely used (at the cost of a thoroughly check, but limited inside the single modules, of all the possible interactions among participants). The inter-modules interactions are instead more controlled, so respecting the decoupling of modules characterising any sound decomposition of systems. It is then possible to prove the properties of Subject Reduction and Session Fidelity for typable modular sessions. Moreover, typability also entails the communication property of Lock Freedom. Typability is shown to be independent from the way a session can be modularised as well as from the order in which the typing of the modules is "sequentialised". We propose, as working example, a modular extension of the above mentioned leader election example of [31].

Overview. In Section 2 we introduce the calculus of multiparty sessions with mixed choice as an extension of the SMPS calculus of [4] and [7]. Modularisable multiparty sessions are then formally presented in Section 3. Global types equipped with a coinductive LTS are defined in Section 4 in the style of [6]. Properties of typable modular sessions are proven in Section 5, namely Subject Reduction, Session Fidelity and Lock Freedom. In that section we also show that typability does not depend on how a session is modularised or on the particular modules considered during typing. A summing-up section, also discussing related and future works, concludes the paper.

2 Multiparty Sessions with Mixed Choice

We present now a SMPS synchronous calculus of multiparty sessions with mixed choice, inspired mainly by [31] and partially by [4]. We assume to have the following denumerable base sets: *messages* (ranged over by $\lambda, \lambda', \ldots$); *session participants* (ranged over by $\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}, \ldots$); *indexes* (ranged over by $i, j, h, k, \ldots$); *finite sets of indexes* (ranged over by $I, J, H, K, \ldots$). We refer to the denumerable set of participant names as $\mathfrak{P}$.

Processes, ranged over by $P, Q, R, S, \ldots$, implement the behaviour of participants. In the following and in later definitions the symbol $::=^{coind}$ does express that the productions have to be interpreted *coinductively* and that only *regular* terms are allowed. Then we can adopt in proofs the coinduction style advocated in [21] which, without any loss of formal rigour, promotes readability and conciseness.

Definition 2.1 (Processes) *i)* Action prefixes *are defined by* $\pi ::= \mathsf{p}?\lambda \mid \mathsf{p}!\lambda$.

ii) Processes *are coinductively defined by*

$$P ::=^{coind} \mathbf{0} \mid \Sigma_{i\in I}\pi_i.P_i$$

where $I \neq \emptyset$ and finite, and $\pi_l = \mathsf{q}?\lambda_l$, $\pi_j = \mathsf{q}?\lambda_j$ (resp. $\pi_l = \mathsf{q}!\lambda_l$, $\pi_j = \mathsf{q}!\lambda_j$) imply $\lambda_l \neq \lambda_j$, for any $l, j \in I$ such that $l \neq j$.

In the above definition, $\Sigma_{i\in I}\pi_i.P_i$ stands, as usual, for the summand of processes $\pi_i.P_i$'s. A $\Sigma_{i\in I}\pi_i.P_i$ process represents the nondeterministic choice of one of the actions π_i, after which the process continues as P_i with $i \in I$. As usual, we assume the summand of processes to be commutative and associative. A prefix π can be any input (i.e. of the form $\mathsf{p}?\lambda$) or output (i.e. of the form $\mathsf{p}!\lambda$) action. We use $\mathbf{0}$ to denote the terminated process. For the sake of readability, we omit trailing $\mathbf{0}$'s in processes.

We define the participants of action prefixes by $\mathsf{prt}(\mathsf{p}?\lambda) = \mathsf{prt}(\mathsf{p}!\lambda) = \{\mathsf{p}\}$. Moreover, we define the participants of processes by

$$\mathsf{prt}(\mathbf{0}) = \emptyset \qquad \mathsf{prt}(\Sigma_{i\in I}\pi_i.P_i) = \bigcup_{i\in I}\mathsf{prt}(\pi_i) \cup \bigcup_{i\in I}\mathsf{prt}(P_i)$$

By the regularity condition and the finiteness of indexes, $\mathsf{prt}(P)$ is finite for any P.

Processes correspond to inductively defined terms of the calculus MCMP (Mixed Choice Multiparty Sessions) as defined in [31], where the μ-operator is used to describe infinite behaviours. Our use of coinductively defined (possibly) infinite terms enables us to get simpler formalisations and proofs with respect to the use of μ-terms. The latter entail just technicalities that can be dealt with as done in the literature on session types and MPST [19], where such terms are usually considered.

Multiparty sessions are parallel compositions of participant-process pairs, where all participants are different.

Definition 2.2 (Multiparty sessions) Multiparty sessions *are defined by* $\mathbb{M} = \mathsf{p}_1[P_1] \parallel \cdots \parallel \mathsf{p}_n[P_n]$ *where $\mathsf{p}_j \neq \mathsf{p}_l$ for $1 \leq j, l \leq n$ and $j \neq l$.*

We assume the standard structural congruence $\equiv$ on multiparty sessions, stating that parallel composition is commutative and associative and has neutral elements $\mathsf{p}[\mathbf{0}]$ for any fresh p. Such a congruence can be inductively defined, as multiparty sessions are. We then write $\mathsf{p}[P] \in \mathbb{M}$ if $\mathbb{M} \equiv \mathsf{p}[P] \parallel \mathbb{M}'$ and $P \neq \mathbf{0}$. Moreover, we define the participants of multiparty sessions by $\mathsf{prt}(\mathbb{M}) = \{\mathsf{p} \mid \mathsf{p}[P] \in \mathbb{M}\}$.

To define the *synchronous operational semantics* of sessions we use an LTS, whose transitions are decorated by *communication labels*, i.e. expressions of the shape $\mathsf{p}\lambda\mathsf{q}$. In the following we use Λ to range over communication labels.

Notation: We use $\pi.P \dotplus Q$ as short for either $\pi.P + Q$ or $\pi.P$. Such a notation enables us to present the

following LTS in a compact and yet formal way, since in our processes – as in [31] – we cannot have unprefixed **0**'s as summands.

Definition 2.3 (LTS for Multiparty Sessions) *The* labelled transition system (LTS) for multiparty sessions *is the closure under structural congruence of the reduction specified by the unique axiom:*

$$[\textsc{Comm}]\ \frac{}{\mathsf{p}[\mathsf{q}!\lambda.P + P'] \parallel \mathsf{q}[\mathsf{p}?\lambda.Q + Q'] \parallel \mathbb{M} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M}}$$

Rule [Comm] makes the communication possible: if participant p is enabled to send message λ to participant q which, in turn, is enabled to receive it, the message can be exchanged. This rule is non-deterministic in the choice of messages exchanged. The implementation issues raised by such operational semantics are similar to those for most of the calculi for concurrency and can be dealt with by resorting to suitable coordination protocols. Such issues are however outside the scope of the present paper.

Note that in the above semantics there is no difference between the behaviours of inputs and outputs, while usually a sender freely chooses among all its available messages. In actual communicating systems, messages would also carry values that are abstracted away in SMPS formalisms for the sake of simplicity. The present calculus (as well as its type system) could however be extended to messages with data.

We define *traces* as (possibly infinite) sequences of communication labels. Formally,

$$\sigma ::=^{coind} \varepsilon \mid \Lambda \cdot \sigma$$

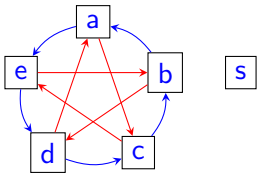
where ε is the empty sequence. When $\sigma = \Lambda_1 \cdot \ldots \cdot \Lambda_n$ $(n \geq 0)$ we write $\mathbb{M} \xrightarrow{\sigma} \mathbb{M}'$ as short for

$$\mathbb{M} \xrightarrow{\Lambda_1} \mathbb{M}_1 \cdots \xrightarrow{\Lambda_n} \mathbb{M}_n = \mathbb{M}'$$

We write $\mathbb{M} \xrightarrow{\sigma}$ with the standard meaning. Moreover, we define the participants of labels by $\mathsf{prt}(\mathsf{p}\lambda\mathsf{q}) = \{\mathsf{p},\mathsf{q}\}$ and the participants of traces – notation $\mathsf{prt}(\sigma)$ – as its obvious extension. We also denote by $\mathscr{L}(\mathbb{M})$ the *set of all labels the session* $\mathbb{M}$ *can emit,* i.e. $\mathscr{L}(\mathbb{M}) = \{\Lambda \mid \mathbb{M} \xrightarrow{\Lambda}\}$.

We present now, in our setting, the leader-election example used in [31] (inspired by [27], in turn inspired by [8]) to make evident the expressive power of mixed choice.

Example 2.4 (Leader election [31]) Five participants $(\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e})$ interact with the aim of electing a leader. Each of them can send to the next participant, in a circular fashion, the message *leader* in order to ask it to become the leader. If such a communication succeeds, the sender terminates. Of course only two of this sort of communications can succeed. The protocol then allows two, among the remaining three participants, to be able to exchange the *leader* message. The receiver is hence considered as the *elect*ed leader and so it informs the station participant s which, in turn, provides to *del*ete the participant which remained inactive during the previous interactions.
The above behaviour is implemented by the following session:



$$\mathbb{E} = \mathsf{a}[P_\mathsf{a}] \parallel \mathsf{b}[P_\mathsf{b}] \parallel \mathsf{c}[P_\mathsf{c}] \parallel \mathsf{d}[P_\mathsf{d}] \parallel \mathsf{e}[P_\mathsf{e}] \parallel \mathsf{s}[P_\mathsf{s}]$$

where
$$P_\mathsf{a} = \mathsf{e}!\mathit{leader} + \mathsf{b}?\mathit{leader}.(\mathsf{c}!\mathit{leader} + \mathsf{d}?\mathit{leader}.\mathsf{s}!\mathit{elect}) + \mathsf{s}?\mathit{del}$$

and (with some abuse of notation)

$$P_\mathsf{s} = \Sigma_{\mathsf{x}\in\{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e}\}}\left(\mathsf{x}?\mathit{elect}.\Sigma_{\mathsf{x}\in\{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e}\}}\mathsf{x}!\mathit{del}\right)$$

Processes of participants b, c, d and e are obtained out of P_a by applying the name substitution $\nu = [\mathsf{a}\mapsto\mathsf{b},\mathsf{b}\mapsto\mathsf{c},\mathsf{c}\mapsto\mathsf{d},\mathsf{d}\mapsto\mathsf{e},\mathsf{e}\mapsto\mathsf{a}]$ as follows:

$$P_\mathsf{b} = P_\mathsf{a}\nu,\ P_\mathsf{c} = P_\mathsf{b}\nu,\ P_\mathsf{d} = P_\mathsf{c}\nu,\ P_\mathsf{e} = P_\mathsf{d}\nu$$

Session $\mathbb{E}$ can be graphically represented by the diagram above on the left, where blue arrows represent the initial possible exchanges of the message *leader* and the red ones the potential further exchanges of such message. The following sequence of reductions is, for instance, the one leading to the election of e:

$$\mathsf{a}\mathit{leader}\mathsf{e}\cdot\mathsf{d}\mathit{leader}\mathsf{c}\cdot\mathsf{c}\mathit{leader}\mathsf{e}\cdot\mathsf{e}\mathit{elect}\mathsf{s}\cdot\mathsf{s}\mathit{del}\mathsf{b} \qquad \diamond$$

Lock Freedom is a relevant property of concurrent systems. We define it in our setting following [26]: roughly, there is always a continuation enabling a participant to communicate whenever it is willing to do so. Lock Freedom entails Deadlock Freedom, since it ensures progress for each participant.

Definition 2.5 (Lock Freedom) *A session* $\mathbb{M}$ *is* lock free *if* $\mathbb{M} \xrightarrow{\sigma} \mathbb{M}'$ *with* σ *finite and* $\mathsf{p} \in \mathsf{prt}(\mathbb{M}')$ *imply* $\mathbb{M}' \xrightarrow{\sigma'\cdot\Lambda}$ *for some* σ' *and* Λ *such that* $\mathsf{p} \notin \mathsf{prt}(\sigma')$ *and* $\mathsf{p} \in \mathsf{prt}(\Lambda)$.

The above definition corresponds to the notion of liveness used in [20] and [22] in a channel-based synchronous communication setting.

3 Modular Multiparty Sessions

"With great [expressive] power comes great responsibility" (Spider-Man's Uncle Ben), since expressive power is often difficult to control and tame. Our aim is to provide a type system for multiparty sessions with mixed choice ensuring, like usual in SMPS, relevant communication properties, together with the guarantee that the overall behaviour of a session faithfully respects what the type assigned to the session, if any, describes. To do that, instead of restricting the expressive power of mixed choice, we decided to consider multiparty sessions that can be – as suggested by a well-known software engineering principle – modularised. A module, in our SMPS setting, is formalised in terms of a subsession inside which participants can freely interact by means of mixed choice. The communications among the modules are instead controlled by imposing them to be performed only by particular participants called "connectors". The processes of the connectors (dubbed connecting processes) must satisfy the restriction that each choice involving a participant not belonging to the module must be between communications with only that participant. Definition 3.1, where $\mathbf{P}$ is the set of module participants, formalises this condition.

Definition 3.1 (Connecting processes) *Given a set of participants* $\mathbf{P}$*, we say that a process* P *is* $\mathbf{P}$-connecting *if for any subprocess of* P*, say* $\Sigma_{i\in I}\pi_i.P_i$*, we have that* $\mathsf{prt}(\pi_j) \notin \mathbf{P}$ *for some* $j \in I$ *implies* $\mathsf{prt}(\pi_i) = \mathsf{prt}(\pi_j)$ *for all* $i \in I$.

A connector is hence a participant of a module (represented by the session $\mathbb{M}$ in the definition below) whose process is a connecting process which can interact with the outside of the module.

Definition 3.2 (Connectors) *Let* $\mathsf{p}[P] \in \mathbb{M}$*. We say that the participant* p *is a* connector *for* $\mathbb{M}$ *if* P *is* $\mathsf{prt}(\mathbb{M})$*-connecting and there is* $\mathsf{q} \in \mathsf{prt}(P)$ *such that* $\mathsf{q} \notin \mathsf{prt}(\mathbb{M})$.

The notions of subsession and session partition are at the basis of that of modular session. We say that $\mathbb{M}'$ is a *subsession* of $\mathbb{M}$ and write $\mathbb{M}' \subseteq \mathbb{M}$, whenever $\mathsf{p}[P] \in \mathbb{M}'$ implies $\mathsf{p}[P] \in \mathbb{M}$. A *partition* of a session $\mathbb{M}$ is, as expected, a set of subsessions $\{\mathbb{M}_h\}_{h\in H}$ of $\mathbb{M}$ such that $\mathsf{prt}(\mathbb{M}) = \bigcup_{h\in H}\mathsf{prt}(\mathbb{M}_h)$ and, for all $h,k \in H$, $\mathsf{prt}(\mathbb{M}_h) \cap \mathsf{prt}(\mathbb{M}_k) = \emptyset$. Therefore, $\mathsf{p}[P] \in \mathbb{M}$ implies that there is a unique $k \in H$ such that $\mathsf{p}[P] \in \mathbb{M}_k$.

Definition 3.3 ($\mathscr{P}$-partition) *Let* $\{\mathbb{M}_h\}_{h\in H}$ *be a partition of a session* $\mathbb{M}$*, and let* $\mathscr{P} = \{\mathbf{P}_k\}_{k\in K}$ *be a partition of a finite subset of the set* $\mathfrak{P}$*. We say that* $\{\mathbb{M}_h\}_{h\in H}$ *is a* $\mathscr{P}$-partition *of* $\mathbb{M}$ *if* $H \subseteq K$ *and* $\mathsf{prt}(\mathbb{M}_h) \subseteq \mathbf{P}_h$ *for all* $h \in H$.

It is not difficult to check that, given a session $\mathbb{M}$ and a partition $\mathscr{P} = \{\mathbf{P}_k\}_{k\in K}$ such that $\mathsf{prt}(\mathbb{M}) \subseteq \bigcup_{k\in K} \mathbf{P}_k$, there is a unique *$\mathscr{P}$-partition* of $\mathbb{M}$.

A session is modularisable (with respect to a partition of participants) when it can be partitioned into subsessions that interact only by means of connectors.

Definition 3.4 ($\mathscr{P}$-modularisation) *A* $\mathscr{P}$-modularisation of $\mathbb{M}$ *is a* $\mathscr{P}$-partition $\{\mathbb{M}_h\}_{h\in H}$ *of* $\mathbb{M}$ *such that, for all* $h \in H$*, the following conditions hold*

i) $\mathsf{p}[P] \in \mathbb{M}_h$ *implies that either* p *is a connector of* $\mathbb{M}_h$ *or, for each* $\mathsf{q} \in \mathsf{prt}(P)$, $\mathsf{q} \in \mathsf{prt}(\mathbb{M})$ *implies* $\mathsf{q} \in \mathsf{prt}(\mathbb{M}_h)$*;*

ii) for each connector $\mathsf{p}[P] \in \mathbb{M}_h$ *and each* $\mathsf{q} \in \mathsf{prt}(P) \setminus \mathsf{prt}(\mathbb{M}_h)$*:*
if $\mathsf{q} \in \mathsf{prt}(\mathbb{M}_k)$*, then* q *is a connector for* $\mathbb{M}_k$

In such a case, we say that $\mathbb{M}$ *is* $\mathscr{P}$*-modularisable.*

It is worth noticing that we impose no limit on the number of connectors present in a module, as well as on the number of external connectors a connector can interact with (see the following example). Besides, a session $\mathbb{M}$ is always $\{\mathsf{prt}(\mathbb{M})\}$-modularisable. For example $\mathbb{M} = \mathsf{p}[\mathsf{q}!\lambda + \mathsf{r}!\lambda']$ is $\{\{\mathsf{p}\}\}$-modularisable and its unique module does not contain any connector. Also, given a partition $\mathscr{P}$ and a session $\mathbb{M}$, there exists a unique $\mathscr{P}$-modularisation of $\mathbb{M}$, if any.

Example 3.5 (Modules with multiple connectors) Let us consider $\mathbb{M} = \mathbb{M}_1 \parallel \mathbb{M}_2$,
where $\mathbb{M}_1 = \mathsf{u}[\mathsf{p}?\lambda.\mathsf{q}!\lambda \ + \ \mathsf{q}!\lambda.\mathsf{p}?\lambda] \parallel \mathsf{p}[\mathsf{u}!\lambda.(\mathsf{r}!\lambda_1 + \mathsf{r}?\lambda_2)] \parallel \mathsf{q}[\mathsf{u}?\lambda.(\mathsf{s}!\lambda_1 + \mathsf{s}?\lambda_2)]$
and $\mathbb{M}_2 = \mathsf{v}[\mathsf{r}!\lambda.\mathsf{s}?\lambda \ + \ \mathsf{s}?\lambda.\mathsf{r}!\lambda] \parallel \mathsf{r}[\mathsf{v}?\lambda.(\mathsf{p}?\lambda_1 + \mathsf{p}!\lambda_2)] \parallel \mathsf{s}[\mathsf{v}!\lambda.(\mathsf{q}?\lambda_1 + \mathsf{q}!\lambda_2)]$.
This session is $\{\{\mathsf{u},\mathsf{p},\mathsf{q}\},\{\mathsf{v},\mathsf{r},\mathsf{s}\}\}$-modularisable and it has multiple connectors. In fact, p and q are the connectors for the module $\{\mathsf{u},\mathsf{p},\mathsf{q}\}$, whereas r and s are the connectors for the module $\{\mathsf{v},\mathsf{r},\mathsf{s}\}$. ◇

In actual programming, refining a modularisation enables to enhance parameters like scalability, maintenance, reusability and many more. In the present setting, it also allows for simpler typings. Any modularisation refinement corresponds to a partition refinement.

Definition 3.6 (Refinement) *A partition* $\mathscr{P}$ *refines a partition* $\mathscr{P}'$ *(notation* $\mathscr{P} \sqsubseteq \mathscr{P}'$*) if* $\mathscr{P} = \{\mathbf{P}_h\}_{h\in H}$*,* $\mathscr{P}' = \{\mathbf{P}'_k\}_{k\in K}$ *and for all* $k \in K$ *there is* $H_k \subseteq H$ *such that* $\mathbf{P}'_k = \bigcup_{h\in H_k} \mathbf{P}_h$*.*

As intuitively evident, it is possible to formally show that coarser partitions maintain modularisability.

Lemma 3.7 *If* $\mathbb{M}$ *is* $\mathscr{P}$*-modularisable and* $\mathscr{P}$ *refines* $\mathscr{P}'$*, then* $\mathbb{M}$ *is* $\mathscr{P}'$*-modularisable.*

Proof. It is not difficult to verify that if conditions (i) and (ii) of Definition 3.4 hold for $\mathscr{P}$, then they hold also for $\mathscr{P}'$. □

From the previous lemma, being $\mathbb{M}$ $\{\mathsf{prt}(\mathbb{M})\}$-modularisable, it immediately follows that $\mathbb{M}$ is $\mathscr{P}$-modularisable for all $\mathscr{P}$ such that $\mathsf{prt}(\mathbb{M}) \subseteq \mathbf{P}$ for some $\mathbf{P} \in \mathscr{P}$.

We can build a minimal refined partition, with respect to $\sqsubseteq$, among those modularising a session. We start with $\mathscr{P}_0 = \{\{\mathsf{p}\} \mid \mathsf{p} \in \mathsf{prt}(\mathbb{M})\}$ and we iteratively build $\mathscr{P}_{i+1}$ by replacing in $\mathscr{P}_i$ the two sets $\mathbf{P},\mathbf{P}'$ with the unique set $\mathbf{P} \cup \mathbf{P}'$ if there is $\mathsf{p}[P] \in \mathbb{M}$ such that $\mathsf{p} \in \mathbf{P}$, $\mathsf{q} \in \mathbf{P}' \cap \mathsf{prt}(P)$ and either P is not $\mathbf{P}$-connecting or q is not a connector for the subsession of $\mathbb{M}$ whose set of participants is $\mathbf{P}'$. It is possible to verify that $\mathbb{M}$ is $\mathscr{P}$-modularisable, where $\mathscr{P}$ is the fixed point of this procedure. We show now such a partition to be also the minimum among the partitions modularising a session.

Lemma 3.8 *The minimal partition modularising a session is unique, i.e. a minimum.*

Proof. Assume toward a contradiction that there are two different minimal (w.r.t $\sqsubseteq$) partitions $\mathscr{P} = \{\mathbf{P}_h\}_{h\in H}$ and $\mathscr{P}' = \{\mathbf{P}'_k\}_{k\in K}$ for modularising a session. This implies that there are $h_1, h_2 \in H$ and $k_0 \in K$ such that $\mathsf{p} \in \mathbf{P}_{h_1}$, $\mathsf{q} \in \mathbf{P}_{h_2}$ and $\{\mathsf{p},\mathsf{q}\} \subseteq \mathbf{P}'_{k_0}$. Then also the partition obtained from $\mathscr{P}'$ by replacing the set $\mathbf{P}'_{k_0}$ with the two sets $\mathbf{P}'_{k_0} \cap \mathbf{P}_{h_1}$ and $\mathbf{P}'_{k_0} \cap \mathbf{P}_{h_2}$ modularises the same session. This is clearly a contradiction. □

It is crucial that $\mathscr{P}$-modularisation is preserved by reduction.

Lemma 3.9 *Let $\mathbb{M}$ be $\mathscr{P}$-modularisable and let $\mathbb{M} \xrightarrow{\Lambda} \mathbb{M}'$. Then also $\mathbb{M}'$ is $\mathscr{P}$-modularisable.*

Proof. Notice that conditions (i) and (ii) of Definition 3.4 are invariant by reduction. In fact a participant p which is a connector in $\mathbb{M}$ is either a connector in $\mathbb{M}'$ too or it has a process whose participants all belong to the subsession containing p in the $\mathscr{P}$-modulation of $\mathbb{M}'$. Therefore a $\mathscr{P}$-modularisation of $\mathbb{M}$ is also a $\mathscr{P}$-modularisation of $\mathbb{M}'$. □

We can notice that, if $\mathscr{P}$ is the minimal partition for modularising $\mathbb{M}$ and $\mathbb{M} \xrightarrow{\Lambda} \mathbb{M}'$, in general $\mathscr{P}$ is not the minimal partition for modularising $\mathbb{M}'$. In fact $\mathsf{prt}(\mathbb{M}')$ can be a proper subset of $\mathsf{prt}(\mathbb{M})$ and a process in $\mathbb{M}$ which is not a connector can reduce to a process which is a connector in $\mathbb{M}'$. For example the minimal partition of $\mathbb{M} \equiv \mathsf{p}[\mathsf{q}!\lambda.\mathsf{r}!\lambda + \mathsf{r}?\lambda'.(\mathsf{q}!\lambda_1 + \mathsf{q}?\lambda_2)] \parallel \mathsf{q}[\mathsf{p}?\lambda + \mathsf{p}?\lambda_1 + \mathsf{p}!\lambda_2] \parallel \mathsf{r}[\mathsf{p}?\lambda + \mathsf{p}!\lambda']$ is $\{\{\mathsf{p},\mathsf{q},\mathsf{r}\}\}$, since p is not a connector. But $\mathbb{M} \xrightarrow{\mathsf{r}\lambda'\mathsf{p}} \mathsf{p}[\mathsf{q}!\lambda_1 + \mathsf{q}?\lambda_2] \parallel \mathsf{q}[\mathsf{p}?\lambda + \mathsf{p}?\lambda_1 + \mathsf{p}!\lambda_2]$ and the minimal partition of this last session is $\{\{\mathsf{p}\},\{\mathsf{q}\}\}$, since p and q are connectors and r disappeared.

Example 3.10 (Modular election) We consider three "local" elections, all managed like in the Example 2.4. The names of the participants of the three local election are like the ones in the Example 2.4, but indexed with indexes in $\{1,2,3\}$. We also consider a further "global election" with participants w_1, w_2 and w_3 and global station gs. Such an election follows a protocol similar to that of the local elections (but simpler, since only three participants do compete for leadership). It can be seen as an election among the winners of the local elections. In the present example the "local leaders", once they are elected, are informed whether they have been elected also "global leader" or not. The above sketched global behaviour is implemented by the following session $\mathbb{E}^{gl}$ made of four subsessions: three local elections $(\mathbb{E}_1, \mathbb{E}_2, \mathbb{E}_3)$ and one global election $\mathbb{G}$ among the "local" winners.

The processes of the participants, are fairly similar to the ones in the Example 2.4 to which a part is added implementing the communication to the local leaders of whether they are also global leader or not.

$$\mathbb{E}^{gl} \quad = \quad \mathbb{E}_1 \parallel \mathbb{E}_2 \parallel \mathbb{E}_3 \parallel \mathbb{G}$$

where, for $1 \le i \le 3$,

$$\mathbb{E}_i \quad = \quad \mathsf{a}_i[P_{\mathsf{a}_i}] \parallel \mathsf{b}_i[P_{\mathsf{b}_i}] \parallel \mathsf{c}_i[P_{\mathsf{c}_i}] \parallel \mathsf{d}_i[P_{\mathsf{d}_i}] \parallel \mathsf{e}_i[P_{\mathsf{e}_i}] \parallel \mathsf{s}_i[P_{\mathsf{s}_i}]$$

with $P_{\mathsf{a}_i} = \mathsf{e}_i!\mathit{leader} + \mathsf{b}_i?\mathit{leader}.\Big(\mathsf{c}_i!\mathit{leader} + \mathsf{d}_i?\mathit{leader}.\mathsf{s}_i!\mathit{elect}.\big(\mathsf{s}_i?\mathit{gleader} + \mathsf{s}_i?\mathit{no}\big)\Big) + \mathsf{s}_i?\mathit{del}$

and $P_{\mathsf{s}_i} = \Sigma_{\mathsf{x}\in\{\mathsf{a}_i,\mathsf{b}_i,\mathsf{c}_i,\mathsf{d}_i,\mathsf{e}_i\}}\mathsf{x}?\mathit{elect}.\big(\mathsf{gs}?\mathit{gleader}.\mathsf{x}!\mathit{gleader}.Q_i + \mathsf{gs}?\mathit{no}.\mathsf{x}!\mathit{no}.Q_i\big)$

with $Q_i = \Sigma_{\mathsf{x}\in\{\mathsf{a}_i,\mathsf{b}_i,\mathsf{c}_i,\mathsf{d}_i,\mathsf{e}_i\}}\mathsf{x}!\mathit{del}$

and with processes of participants b_i, c_i, d_i and e_i obtained out of P_{a_i} by applying the name substitution

$\nu_i = [\mathsf{a}_i \mapsto \mathsf{b}_i, \mathsf{b}_i \mapsto \mathsf{c}_i, \mathsf{c}_i \mapsto \mathsf{d}_i, \mathsf{d}_i \mapsto \mathsf{e}_i, \mathsf{e}_i \mapsto \mathsf{a}_i]$ as in Example 2.4

and

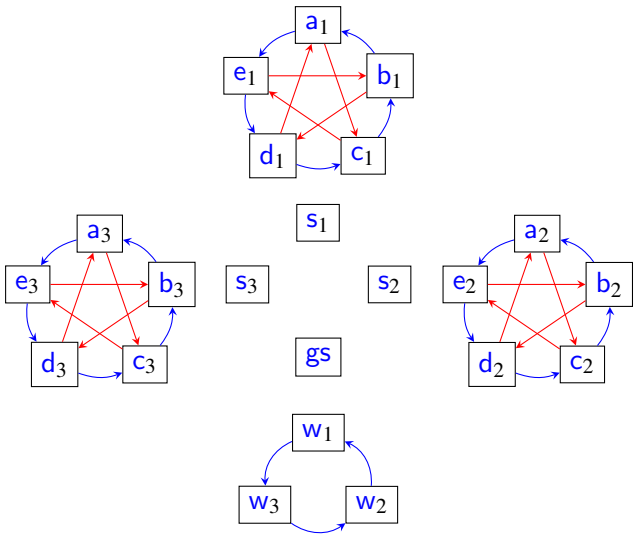


Figure 1: The three local elections and the global one of Example 3.10.

$$\mathbb{G} \;=\; \mathsf{w}_1[P_{\mathsf{w}_1}] \parallel \mathsf{w}_2[P_{\mathsf{w}_2}] \parallel \mathsf{w}_3[P_{\mathsf{w}_3}] \parallel \mathsf{gs}[P_{\mathsf{gs}}]$$

with $P_{\mathsf{w}_i} = \mathsf{w}_{i+2}!\mathit{leader} + \mathsf{w}_{i+1}?\mathit{leader}.\mathsf{gs}!\mathit{gleader} + \mathsf{gs}?\mathit{del}$

and $P_{\mathsf{gs}} = \Sigma_{i\in\{1,2,3\}}\mathsf{w}_i?\mathit{gleader}.\mathsf{s}_i!\mathit{gleader}.\mathsf{s}_{i+1}!\mathit{no}.\mathsf{s}_{i+2}!\mathit{no}.\big(\Sigma_{i\in\{1,2,3\}}\mathsf{w}_i!\mathit{del}\big)$
where all the indexes above have to be considered modulo 3, plus 1.

It is not difficult to check that $\mathbb{E}^{gl}$ is $\mathscr{P}$-modularisable for $\mathscr{P} = \{\mathsf{prt}(\mathbb{E}_1), \mathsf{prt}(\mathbb{E}_2), \mathsf{prt}(\mathbb{E}_3), \mathsf{prt}(\mathbb{G})\}$, where $\mathsf{s}_1, \mathsf{s}_2, \mathsf{s}_3$ and gs are the connectors for, respectively, the modules $\mathbb{E}_1$, $\mathbb{E}_2$, $\mathbb{E}_3$ and $\mathbb{G}$. ◇

4 A Type System for Modular Sessions

Global types are used to represent the overall behaviour of multiparty sessions. Here we use a notion of global type similar to the one in MPST but, following SMPS, we define them coinductively, as possibly infinite regular terms.

Definition 4.1 (Global types) Global types *are coinductively defined by:*

$$\mathsf{G} ::=^{coind} \mathtt{End} \mid \Sigma_{i\in I}\Lambda_i.\mathsf{G}_i$$

where $I \neq \emptyset$ and finite, and for any $j,l \in I$ such that $j \neq l$, $\Lambda_i = \mathsf{p}\lambda_i\mathsf{q}$ and $\Lambda_j = \mathsf{p}\lambda_j\mathsf{q}$ imply $\lambda_j \neq \lambda_l$.

We define the participants of global types by $\mathsf{prt}(\mathtt{End}) = \emptyset$ and $\mathsf{prt}(\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i) = \bigcup_{i\in I}\mathsf{prt}(\Lambda_i) \cup \bigcup_{i\in I}\mathsf{prt}(\mathsf{G}_i)$. By the regularity condition, $\mathsf{prt}(\mathsf{G})$ is finite for any G. As usual, trailing $\mathtt{End}$'s will be omitted.

As mentioned in the Introduction, in standard SMPS, typing rules take into account single interactions between pairs of participants. In our mixed-choice setting, that approach would allow to type only sessions whose independent parts were intrinsically sequential, so ruling out protocols like the leader election and the modular election. We hence consider a typing rule where all the interactions involving

the participants of a single module $\widehat{\mathbb{M}}$ (so including also the communications of the connectors of $\widehat{\mathbb{M}}$ with other modules) are taken into account. Such a set of interactions is formalised through the notion of coherent set of communication labels. A module $\widehat{\mathbb{M}}$ is therefore indirectly represented in the rule in terms of its corresponding coherent set and referred to, when necessary, as the *witness* of such a set.

Definition 4.2 (Coherent set of communication labels) *A set of labels $\{\Lambda_i\}_{i\in I}$ is* $\mathscr{P}$-coherent for $\mathbb{M}$ *if $\mathbb{M}$ is $\mathscr{P}$-modularisable and there exists an element $\widehat{\mathbb{M}}$ of the (unique) $\mathscr{P}$-modularisation of $\mathbb{M}$ such that*

$$\{\Lambda_i\}_{i\in I} = \{\Lambda \in \mathscr{L}(\mathbb{M}) \mid \mathsf{prt}(\Lambda) \cap \mathsf{prt}(\widehat{\mathbb{M}}) \neq \emptyset\}$$

The $\widehat{\mathbb{M}}$ above is called witness *for the $\mathscr{P}$-coherence of $\{\Lambda_i\}_{i\in I}$.*

Example 4.3 (Witness) Let us consider $\mathbb{M}'_1 = \mathsf{u}[\mathsf{q}!\lambda] \parallel \mathsf{p}[\mathsf{r}!\lambda_1 + \mathsf{r}?\lambda_2] \parallel \mathsf{q}[\mathsf{u}?\lambda.(\mathsf{s}!\lambda_1 + \mathsf{s}?\lambda_2)]$ and $\mathbb{M}_2 = \mathsf{v}[\mathsf{r}!\lambda.\mathsf{s}?\lambda \ + \ \mathsf{s}?\lambda.\mathsf{r}!\lambda] \parallel \mathsf{r}[\mathsf{v}?\lambda.(\mathsf{p}?\lambda_1 + \mathsf{p}!\lambda_2)] \parallel \mathsf{s}[\mathsf{v}!\lambda.(\mathsf{q}?\lambda_1 + \mathsf{q}!\lambda_2)]$. The session $\mathbb{M}'_1 \parallel \mathbb{M}_2$ can be obtained by reducing the session of Example 3.5. It is still $\{\{\mathsf{u},\mathsf{p},\mathsf{q}\},\{\mathsf{v},\mathsf{r},\mathsf{s}\}\}$-modularisable and $\mathbb{M}'_1$ is the witness for the $\{\{\mathsf{u},\mathsf{p},\mathsf{q}\},\{\mathsf{v},\mathsf{r},\mathsf{s}\}\}$-coherent set of labels $\{\mathsf{u}\lambda\mathsf{q},\ \mathsf{p}\lambda_1\mathsf{r},\ \mathsf{r}\lambda_2\mathsf{p}\}$. ◇

Here and in the following, the double line indicates that the rules are interpreted coinductively.

Definition 4.4 (Type system) *The type system $\vdash^{\mathscr{P}}$ is defined by the following axiom and rule, where sessions are considered modulo structural congruence:*

$$[\text{End}]\ \frac{}{\mathtt{End} \vdash^{\mathscr{P}} \mathsf{p}[\mathbf{0}]}$$

$$[\text{TComm}]\ \frac{\begin{array}{c}\mathbb{M} \xrightarrow{\Lambda_i} \mathbb{M}_i \qquad \mathsf{G}_i \vdash^{\mathscr{P}} \mathbb{M}_i \qquad \forall i \in I \neq \emptyset \\ \{\Lambda_i\}_{i\in I} \textit{ is } \mathscr{P}\textit{-coherent for } \mathbb{M} \qquad \mathsf{prt}(\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i) = \mathsf{prt}(\mathbb{M})\end{array}}{\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i \vdash^{\mathscr{P}} \mathbb{M}}$$

It is not difficult to check that we can derive the global type $\mathsf{u}\lambda\mathsf{q}.\mathsf{p}\lambda\mathsf{u}.\mathsf{G}_1 + \mathsf{p}\lambda\mathsf{u}.\mathsf{u}\lambda\mathsf{q}.\mathsf{G}_1$, where $\mathsf{G}_1 = \mathsf{v}\lambda\mathsf{r}.\mathsf{s}\lambda\mathsf{v}.\mathsf{G}_2 + \mathsf{s}\lambda\mathsf{v}.\mathsf{v}\lambda\mathsf{r}.\mathsf{G}_2$, $\mathsf{G}_2 = \mathsf{p}\lambda_1\mathsf{r}.\mathsf{G}_3 + \mathsf{r}\lambda_2\mathsf{p}.\mathsf{G}_3$, and $\mathsf{G}_3 = \mathsf{q}\lambda_1\mathsf{s} + \mathsf{s}\lambda_2\mathsf{q}$, for the session of Example 3.5.

The condition "$\{\Lambda_i\}_{i\in I}$ is $\mathscr{P}$-coherent for $\mathbb{M}$" is essential to get Subject Reduction. In fact, by allowing any subset of $\mathscr{L}(\mathbb{M})$ as $\{\Lambda_i\}_{i\in I}$ in [TComm], we could derive $\mathsf{p}\lambda'\mathsf{r}.\mathsf{p}\lambda\mathsf{q} \vdash^{\mathscr{P}} \mathbb{M}_0$ for

$$\mathbb{M}_0 \equiv \mathsf{p}[\mathsf{q}!\lambda + \mathsf{r}!\lambda'.\mathsf{q}!\lambda] \parallel \mathsf{q}[\mathsf{p}?\lambda] \parallel \mathsf{r}[\mathsf{p}?\lambda']$$

regardless of $\mathscr{P}$. However, we would also have $\mathbb{M}_0 \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{r}[\mathsf{p}?\lambda']$, with $\mathsf{r}[\mathsf{p}?\lambda']$ untypable. The above example also shows that, in order to get Subject Reduction it is necessary that, at any moment, a connector can interact with one other connector only. Let us assume to relax Definition 3.1 as follows

> *Given a set of participants* $\mathbf{P}$*, we say that a process P is* $\mathbf{P}$-connecting *if for any subprocess of P, say $\Sigma_{i\in I}\pi_i.P_i$, we have that $\mathsf{prt}(\pi_j) \notin \mathbf{P}$ for some $j \in I$ implies $\mathsf{prt}(\pi_i) \notin \mathbf{P}$ for all $i \in I$.*

This would imply $\mathbb{M}_0$ above to be $\{\{\mathsf{p}\},\{\mathsf{q}\},\{\mathsf{r}\}\}$-modularisable and all the participants would turn out to be connectors in their respective modules. Hence $\mathbb{M}_0$ would be $\vdash^{\{\{\mathsf{p}\},\{\mathsf{q}\},\{\mathsf{r}\}\}}$ typable, whereas $\mathsf{r}[\mathsf{p}?\lambda']$ would not.

The condition "$\mathsf{prt}(\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i) = \mathsf{prt}(\mathbb{M})$" is necessary to get Lock Freedom. For example without this condition we could derive $\mathsf{G} \vdash^{\{\{\mathsf{p}\},\{\mathsf{q}\},\{\mathsf{r}\}\}} \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathsf{r}[\mathsf{s}!\lambda]$ with $P = \mathsf{q}!\lambda.P$, $Q = \mathsf{p}?\lambda.Q$ and $\mathsf{G} = \mathsf{p}\lambda\mathsf{q}.\mathsf{G}$. For what concerns the condition $I \neq \emptyset$, let us consider one of our previous examples, namely $\mathbb{M} = \mathsf{p}[\mathsf{q}!\lambda + \mathsf{r}!\lambda']$. Such a session can be uniquely modularised with itself as possible module and it is not lock free. In fact it is not typable, since its unique coherent set is empty.

It can be proved that $\mathscr{P}$-coherence of a label set for a session is preserved by reducing the session with a label not belonging to the $\mathscr{P}$-coherent set, see Lemma 5.6.

The type system is decidable, since processes and global types are regular, and there is only a finite number of partitions for the participants of a session. More interesting, typability of a session does depend on the choice neither of the $\mathscr{P}$-coherent sets nor of the partition $\mathscr{P}$ (see, respectively, Theorems 5.11 and 5.12).

We define the semantics of global types via a coinductive formal system, as done first in [6]. Such a coinductive definition enables to take into account global types containing branches, where some communications can be indefinitely procrastinated, see Example 4.7. In order to do that, it is handy to associate to a global type the set of communication labels which might (not necessarily) decorate its transitions. We dub them capabilities of the global type.

Definition 4.5 (Capabilities) *Capabilities of global types are defined by:*

$$cap(\mathtt{End}) = \emptyset \quad cap(\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i) = \{\Lambda_i\}_{i\in I} \cup \bigcup_{i\in I} cap(\mathsf{G}_i)$$

Definition 4.6 (LTS for global types) *The* labelled transition system (LTS) for global types *is specified by the following axiom and rule:*

$$[\text{E-COMM}]\ \frac{}{\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i \xrightarrow{\Lambda_j} \mathsf{G}_j} \qquad j \in I$$

$$[\text{I-COMM}]\ \frac{\mathsf{G}_i \xrightarrow{\Lambda} \mathsf{G}_i' \quad \Lambda \in cap(\mathsf{G}_i) \quad \mathsf{prt}(\Lambda) \cap \mathsf{prt}(\Lambda_i) = \emptyset \quad \forall i \in I}{\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i \xrightarrow{\Lambda} \Sigma_{i\in I}\Lambda_i.\mathsf{G}_i'}$$

Axiom [E-COMM] formalises the fact that, in a session exposing the behaviour $\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i$, the communication labelled Λ_j for any $j \in I$ can happen. When such a communication is actually performed, the resulting session will expose the behaviour G_j.

Rule [I-COMM] enables to describe independent and concurrent communications, even if global types apparently look like sequential descriptions of sessions' overall behaviours. In fact, behaviours involving participants ready to interact with each other uniformly in all branches of a global type, can do that if neither of them is involved in an interaction appearing at top level in the global type. The condition $\Lambda \in cap(\mathsf{G}_i)$ in Rule [I-COMM] is needed because such a rule is coinductive. In fact, without such a condition, we could get the following infinite derivation for $\mathsf{G} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{G}$ with $\mathsf{G} = \mathsf{r}\lambda'\mathsf{s}.\mathsf{G}$:

$$\mathscr{D} = \frac{\mathscr{D}}{\mathsf{G} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{G}}\ [\text{I-COMM}]$$

Example 4.7 (Use of coinduction in Rule [I-COMM]) As shown in [6] the coinductive formulation of Rule [I-COMM] allows to get $\mathsf{G} \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{G}'$, where $\mathsf{G} = \mathsf{r}\lambda_1\mathsf{s}.\mathsf{G} + \mathsf{r}\lambda_2\mathsf{s}.\mathsf{p}\lambda\mathsf{q}$ and $\mathsf{G}' = \mathsf{r}\lambda_1\mathsf{s}.\mathsf{G}' + \mathsf{r}\lambda_2\mathsf{s}$. The inductive definition of this rule does not allow the shown transition. ◇

Example 4.8 (Typing the modular election session) In Figure 2 we provide a typing for the modular session $\mathbb{E}^{gl}$ of Example 3.10 in the type system $\vdash^{\mathscr{P}}$ with

$$\mathscr{P} = \bigcup_{i\in\{1,2,3\}}\{\{\mathsf{a}_i,\mathsf{b}_i,\mathsf{c}_i,\mathsf{d}_i,\mathsf{e}_i,\mathsf{s}_i\}\} \cup \{\{\mathsf{w}_1,\mathsf{w}_2,\mathsf{w}_3,\mathsf{gs}\}\}$$

We use colours for representing reduced participants. In particular, a participant has the colour
□ in its initial state; ■ after one interaction; ■ after two interactions;
■ after three interactions; ■ after four interactions; ■ when terminated.

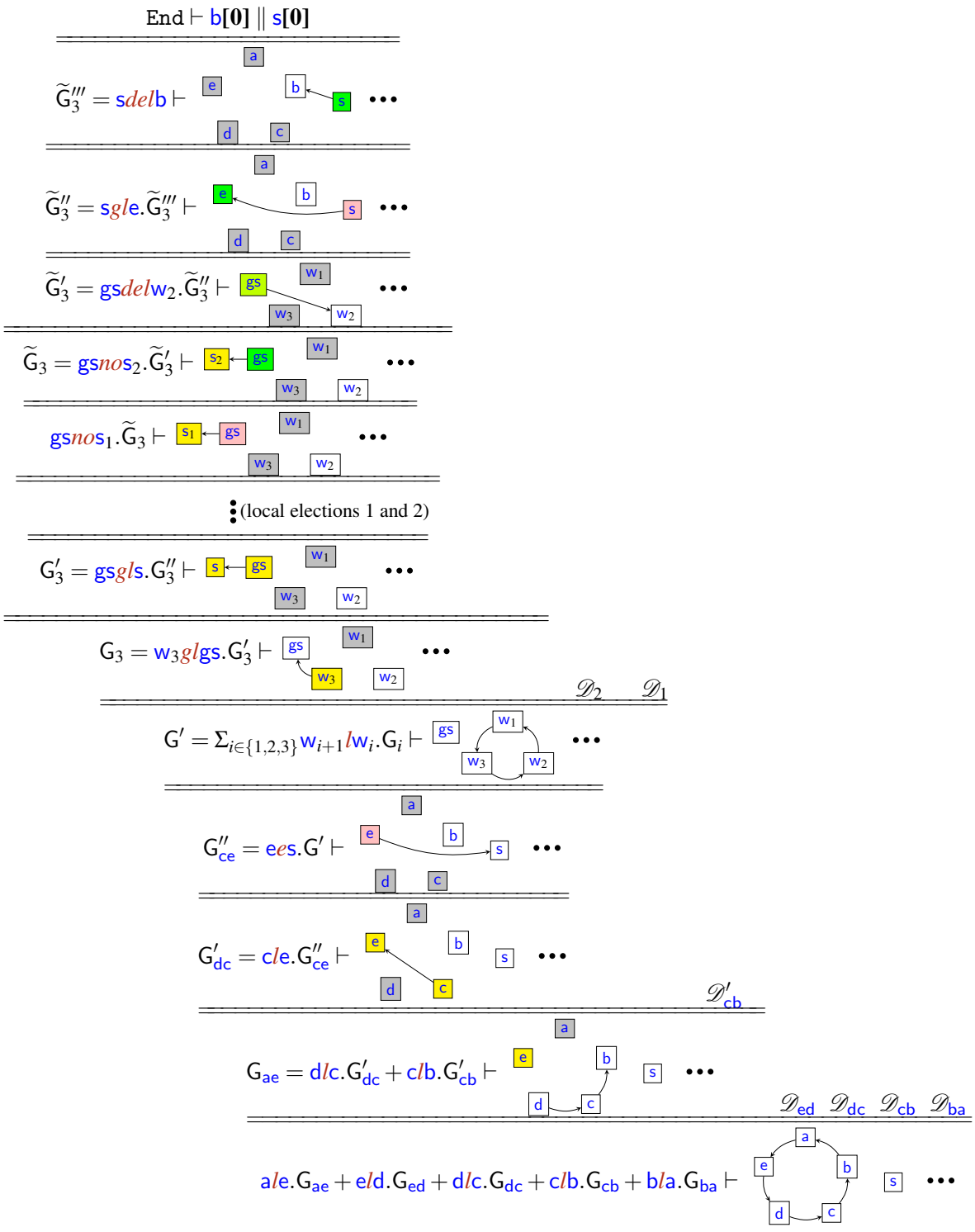


Figure 2: A type derivation for Example 3.10.

For instance:
$\boxed{\mathsf{gs}} = \mathsf{gs}[\Sigma_{i\in\{1,2,3\}}\mathsf{w}_i?gleader.\mathsf{s}_i!gleader.\mathsf{s}_{i+1}!no.\mathsf{s}_{i+2}!no.\big(\Sigma_{i\in\{1,2,3\}}\mathsf{w}_i!del\big)]$
$\boxed{\mathsf{gs}} = \mathsf{gs}[\mathsf{s}_3!gleader.\mathsf{s}_1!no.\mathsf{s}_2!no.\big(\Sigma_{i\in\{1,2,3\}}\mathsf{w}_i!del\big)]$ $\boxed{\mathsf{gs}} = \mathsf{gs}[\mathsf{s}_1!no.\mathsf{s}_2!no.\big(\Sigma_{i\in\{1,2,3\}}\mathsf{w}_i!del\big)]$
$\boxed{\mathsf{gs}} = \mathsf{gs}[\mathsf{s}_2!no.\big(\Sigma_{i\in\{1,2,3\}}\mathsf{w}_i!del\big)]$ $\boxed{\mathsf{gs}} = \mathsf{gs}[\Sigma_{i\in\{1,2,3\}}\mathsf{w}_i!del]$ $\boxed{\mathsf{gs}} = \mathsf{gs}[\mathbf{0}]$

In Figure 2, arrows do connect pair of participants forming a redex. Moreover, in the conclusion of the rules we show only the participants of the module containing the coherent set of reductions and, in case, the "external" connectors. The rest of the session will be denoted by " $\bullet\bullet\bullet$ ". Also, the figure shows only one branch of the typing derivation tree: the one concerning the global election of e_3.

For the sake of readability, *l* and *gl* are abbreviation for, respectively, *leader* and *gleader*. Moreover, $\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e}$ and s stand for $\mathsf{a}_3,\mathsf{b}_3,\mathsf{c}_3,\mathsf{d}_3,\mathsf{e}_3$ and s_3. ◇

5 Properties

A subsession of the shape $\mathsf{p}[\mathsf{q}!\lambda.P \dotplus P']\parallel \mathsf{q}[\mathsf{p}?\lambda.Q \dotplus Q']$ is called a *redex* and $\mathsf{p}[P]\parallel\mathsf{q}[Q]$ is the *contractum* of the redex. In a transition labelled by $\mathsf{p}\lambda\mathsf{q}$ both the redex and the contractum are uniquely determined.

Lemma 5.1 *If* $\mathbb{M}\xrightarrow{\mathsf{p}\lambda\mathsf{q}}\mathbb{M}'$, *then there exists a unique redex* $\mathsf{p}[\mathsf{q}!\lambda.P \dotplus P']\parallel \mathsf{q}[\mathsf{p}?\lambda.Q \dotplus Q']$ *such that*

$$\mathbb{M}\equiv \mathsf{p}[\mathsf{q}!\lambda.P \dotplus P']\parallel \mathsf{q}[\mathsf{p}?\lambda.Q \dotplus Q']\parallel\mathbb{M}''$$

and $\mathbb{M}'\equiv\mathsf{p}[P]\parallel\mathsf{q}[Q]\parallel\mathbb{M}''$.

Proof. Immediate by the definition of session LTS. □

Rule [Comm] in Definition 2.3 entails an easy relation between the participants connected by reductions in a session.

Lemma 5.2 *If* $\mathbb{M}\xrightarrow{\Lambda}\mathbb{M}'$, *then* $\mathsf{prt}(\mathbb{M})=\mathsf{prt}(\Lambda)\cup\mathsf{prt}(\mathbb{M}')$.

It is not difficult to check that the participants of a session and of its global type are the same.

Lemma 5.3 *If* $\mathsf{G}\vdash^{\mathscr{P}}\mathbb{M}$, *then* $\mathsf{prt}(\mathsf{G})=\mathsf{prt}(\mathbb{M})$.

The following technical lemma relating capabilities and possible reductions of a global type will be handy later on.

Lemma 5.4 *If* $\mathsf{G}\xrightarrow{\Lambda}\mathsf{G}'$, *then* $\Lambda\in cap(\mathsf{G})$.

Proof. By cases on the applied axiom/rule justifying $\mathsf{G}\xrightarrow{\Lambda}\mathsf{G}'$. If this is [E-Comm], then $\mathsf{G}=\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i$ and $\Lambda=\Lambda_j$ for some $j\in I$ and $\Lambda_j\in cap(\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i)$ by Definition 4.5.

Otherwise, $\mathsf{G}=\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i$ and $\mathsf{G}'=\Sigma_{i\in I}\Lambda_i.\mathsf{G}'_i$ by Rule [I-Comm], where $\mathsf{G}_i\xrightarrow{\Lambda}\mathsf{G}'_i$, $\Lambda\in cap(\mathsf{G}_i)$ and $\mathsf{prt}(\Lambda)\cap\mathsf{prt}(\Lambda_i)=\emptyset$ for all $i\in I$. This implies $\Lambda\in cap(\mathsf{G})$, since $\bigcup_{i\in I}cap(\mathsf{G}_i)\subseteq cap(\mathsf{G})$ by Definition 4.5. □

For showing Subject Reduction it is crucial to ensure that the $\mathscr{P}$-coherence of a set of labels is preserved by reducing a label not belonging to this set, see Lemma 5.6 whose proof uses Lemma 5.5 below.

Lemma 5.5 *Let* $\{\Lambda_i\}_{i\in I}$ *be* $\mathscr{P}$*-coherent for* $\mathbb{M}$ *and let* $\Lambda\in\mathscr{L}(\mathbb{M})$. *Moreover, let* $\Lambda\neq\Lambda_i$ *for all* $i\in I$. *Then* $\mathsf{prt}(\Lambda)\cap\mathsf{prt}(\Lambda_i)=\emptyset$ *for all* $i\in I$.

Proof. By definition of coherence (Definition 4.2), we have a subsession $\widehat{\mathbb{M}}$ of $\mathbb{M}$ witnessing the $\mathscr{P}$-coherence of $\{\Lambda_i\}_{i\in I}$. By contradiction, let us assume $\mathsf{prt}(\Lambda) \cap \mathsf{prt}(\Lambda_j) \neq \emptyset$ for some $j \in I$. By definition of $\mathscr{P}$-modularisation, this implies that $\mathsf{prt}(\Lambda_j)$ contains a connector p of $\widehat{\mathbb{M}}$. Let $\mathsf{prt}(\Lambda_j) = \{\mathsf{p},\mathsf{q}\}$ with $\mathsf{q} \notin \mathsf{prt}(\widehat{\mathbb{M}})$ and $\mathsf{prt}(\Lambda) = \{\mathsf{q},\mathsf{r}\}$. Then $\mathsf{q} \in \mathsf{prt}(\mathbb{M})$ and the process Q of q must have a choice between a communication with p and a communication with r. But this is impossible, since q must be a connector for some subsession $\widehat{\mathbb{M}}'$ of $\mathbb{M}$ by condition (ii) of Definition 3.4 and then the process Q must be $\mathsf{prt}(\widehat{\mathbb{M}}')$-connecting by Definition 3.2. □

Lemma 5.6 (Coherence preservation) *Let $\{\Lambda_i\}_{i\in I}$ be $\mathscr{P}$-coherent for $\mathbb{M}$ and let $\mathbb{M} \xrightarrow{\Lambda} \mathbb{M}'$. Then $\Lambda \notin \{\Lambda_i\}_{i\in I}$ implies that $\{\Lambda_i\}_{i\in I}$ is $\mathscr{P}$-coherent for $\mathbb{M}'$ as well.*

Proof. Let $\widehat{\mathbb{M}}$ be the subsession of $\mathbb{M}$ witnessing the $\mathscr{P}$-coherence of $\{\Lambda_i\}_{i\in I}$ for $\mathbb{M}$. From $\Lambda \notin \{\Lambda_i\}_{i\in I}$ and Lemma 5.5 we get that $\mathsf{prt}(\Lambda) \cap \mathsf{prt}(\Lambda_i) = \emptyset$ for all $i \in I$. This implies that the reduction $\xrightarrow{\Lambda}$ cannot affect any reduction with label in $\{\Lambda_i\}_{i\in I}$. Hence $\widehat{\mathbb{M}}$ is a witness of the $\mathscr{P}$-coherence of $\{\Lambda_i\}_{i\in I}$ also for $\mathbb{M}'$. □

Notice how the conditions on connectors (Definitions 3.1 and 3.2) are crucial in getting the property that $\mathsf{prt}(\Lambda) \cap \mathsf{prt}(\Lambda_i) = \emptyset$ for all $i \in I$ in the above result of coherence preservation. If we allowed connectors to communicate with external partners having unrestricted processes we could consider the $\{\{\mathsf{p}\},\{\mathsf{r},\mathsf{s}\}\}$-modularisation of $\mathbb{M} = \mathsf{p}[\mathsf{r}!\lambda] \parallel \mathsf{r}[\mathsf{p}?\lambda + \mathsf{s}?\lambda] \parallel \mathsf{s}[\mathsf{r}!\lambda]$. In such a case, the set $\{\mathsf{p}\lambda\mathsf{r}\}$ would be $\{\{\mathsf{p}\},\{\mathsf{r},\mathsf{s}\}\}$-coherent with witness $\mathbb{M}' = \mathsf{p}[\mathsf{r}!\lambda]$. However, we would also have that $\mathbb{M} \xrightarrow{\mathsf{s}\lambda\mathsf{r}} \mathbb{M}'$, but $\{\mathsf{p}\lambda\mathsf{r}\}$ would not be $\{\{\mathsf{p}\},\{\mathsf{r},\mathsf{s}\}\}$-coherent for $\mathbb{M}'$, since $\mathsf{p}\lambda\mathsf{r} \notin \mathscr{L}(\mathbb{M}')$.

Theorem 5.7 (Subject Reduction) *If $\mathsf{G} \vdash^{\mathscr{P}} \mathbb{M}$ and $\mathbb{M} \xrightarrow{\Lambda} \mathbb{M}'$, then $\mathsf{G}' \vdash^{\mathscr{P}} \mathbb{M}'$ and $\mathsf{G} \xrightarrow{\Lambda} \mathsf{G}'$ for some G'.*

Proof. By coinduction on the derivation of $\mathsf{G} \vdash^{\mathscr{P}} \mathbb{M}$. Let $\Lambda = \mathsf{p}\lambda\mathsf{q}$. By Lemma 5.1, if $\mathbb{M} \xrightarrow{\Lambda} \mathbb{M}'$, then there exists a unique redex

$$\mathscr{R} = \mathsf{p}[\mathsf{q}!\lambda.P \dddot{+} P'] \parallel \mathsf{q}[\mathsf{p}?\lambda.Q \dddot{+} Q']$$

such that $\mathbb{M} \equiv \mathscr{R} \parallel \mathbb{M}''$ and $\mathbb{M}' \equiv \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M}''$ for some $\mathbb{M}''$. By the hypothesis that $\mathsf{G} \vdash^{\mathscr{P}} \mathbb{M}$ we know that G is of the form $\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i$ and the derivation ends by

$$[\text{TCOMM}]\ \frac{\mathbb{M} \xrightarrow{\Lambda_i} \mathbb{M}_i \qquad \mathsf{G}_i \vdash^{\mathscr{P}} \mathbb{M}_i \qquad \forall i \in I \neq \emptyset \qquad \{\Lambda_i\}_{i\in I} \text{ is } \mathscr{P}\text{-coherent for } \mathbb{M} \qquad \mathsf{prt}(\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i) = \mathsf{prt}(\mathbb{M})}{\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i \vdash^{\mathscr{P}} \mathbb{M}}$$

We proceed by distinguishing the two possible following cases.

Case $\mathsf{p}\lambda\mathsf{q} = \Lambda_j$ for some $j \in I$. By the premises of the rule, we have $\mathbb{M} \xrightarrow{\Lambda} \mathbb{M}_j$ and $\mathsf{G}_j \vdash^{\mathscr{P}} \mathbb{M}_j$, where $\mathbb{M}' = \mathbb{M}_j$. Moreover, it immediately follows that $\mathsf{G} \xrightarrow{\Lambda} \mathsf{G}_j$ by Axiom [E-COMM].

Case $\mathsf{p}\lambda\mathsf{q} \neq \Lambda_i$ for all $i \in I$. We have that, for all $i \in I$, $\mathbb{M}_i \equiv \mathscr{R} \parallel \mathbb{M}'_i$ for some $\mathbb{M}'_i$. Hence we get that $\mathbb{M}_i \xrightarrow{\Lambda} \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M}'_i$ for all $i \in I$. Moreover, for all $i \in I$, $\mathbb{M}'' \xrightarrow{\Lambda_i} \mathbb{M}'_i$. By the coinduction hypothesis on the premises of the rule, we have that, for all $i \in I$, $\mathsf{G}'_i \vdash^{\mathscr{P}} \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M}'_i$ and $\mathsf{G}_i \xrightarrow{\Lambda} \mathsf{G}'_i$ for some G'_i. Now, by Lemma 5.4, we get that $\Lambda \in cap(\mathsf{G}_i)$ for all $i \in I$, hence we have that $\mathsf{G} \xrightarrow{\Lambda} \Sigma_{i\in I}\Lambda_i.\mathsf{G}'_i$ by Rule [I-COMM]. From $\mathbb{M} \xrightarrow{\Lambda} \mathbb{M}' \equiv \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M}''$ and $\mathbb{M}'' \xrightarrow{\Lambda_i} \mathbb{M}'_i$ we get $\mathbb{M}' \xrightarrow{\Lambda_i} \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M}'_i$ for all $i \in I$, which imply, by Lemma 5.2,

$$\mathsf{prt}(\mathbb{M}') = \bigcup_{i\in I} \mathsf{prt}(\Lambda_i) \cup \bigcup_{i\in I} \mathsf{prt}(\mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M}'_i)$$

By Lemma 5.3, $\mathsf{G}'_i \vdash^{\mathscr{P}} \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M}'_i$ gives $\mathsf{prt}(\mathsf{G}'_i) = \mathsf{prt}(\mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M}'_i)$ for all $i \in I$. Hence $\mathsf{prt}(\Sigma_{i\in I}\Lambda_i.\mathsf{G}'_i) = \bigcup_{i\in I}\mathsf{prt}(\Lambda_i) \cup \bigcup_{i\in I}\mathsf{prt}(\mathsf{G}'_i) = \mathsf{prt}(\mathbb{M}')$. Moreover, from Lemma 5.6 and $\mathsf{p}\lambda\mathsf{q} \neq \Lambda_i$ for all $i \in I$, it follows that $\{\Lambda_i\}_{i\in I}$ is $\mathscr{P}$-coherent for $\mathbb{M}'$ as well. Therefore Rule [TComm] applies, namely

$$\frac{\mathbb{M}' \xrightarrow{\Lambda_i} \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M}'_i \quad \mathsf{G}'_i \vdash^{\mathscr{P}} \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M}'_i \quad \forall i \in I \neq \emptyset \qquad \{\Lambda_i\}_{i\in I} \text{ is } \mathscr{P}\text{-coherent for } \mathbb{M}' \quad \mathsf{prt}(\Sigma_{i\in I}\Lambda_i.\mathsf{G}'_i) = \mathsf{prt}(\mathbb{M}')}{\Sigma_{i\in I}\Lambda_i.\mathsf{G}'_i \vdash^{\mathscr{P}} \mathbb{M}'}$$

□

Theorem 5.8 (Session Fidelity) *If* $\mathsf{G} \vdash \mathbb{M}$ *and* $\mathsf{G} \xrightarrow{\Lambda} \mathsf{G}'$, *then* $\mathbb{M} \xrightarrow{\Lambda} \mathbb{M}'$ *and* $\mathsf{G}' \vdash \mathbb{M}'$ *for some* $\mathbb{M}'$.

Proof. By coinduction on the derivation of $\mathsf{G} \xrightarrow{\Lambda} \mathsf{G}'$. We distinguish two cases according to the axiom/rule justifying $\mathsf{G} \xrightarrow{\Lambda} \mathsf{G}'$.

Axiom [E-Comm]: then $\mathsf{G} = \Sigma_{i\in I}\Lambda_i.\mathsf{G}_i$, $\Lambda = \Lambda_j$ and $\mathsf{G}' = \mathsf{G}_j$ for some $j \in I$. Since $\mathsf{G} \neq \mathtt{End}$, the last rule in the derivation of $\mathsf{G} \vdash \mathbb{M}$ must be [TComm], which implies that $\Lambda = \mathsf{p}\lambda\mathsf{q}$ for some p, λ and q such that

$$\mathbb{M} \equiv \mathsf{p}[\mathsf{q}!\lambda.P \dotplus P'] \parallel \mathsf{q}[\mathsf{p}?\lambda.Q \dotplus Q'] \parallel \mathbb{M}_0 \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{p}[P] \parallel \mathsf{q}[Q] \parallel \mathbb{M}_0 \equiv \mathbb{M}'$$

for some $\mathbb{M}_0$, and $\mathsf{G}' \vdash^{\mathscr{P}} \mathbb{M}'$.

Rule [I-Comm]: then $\mathsf{G} = \Sigma_{i\in I}\Lambda_i.\mathsf{G}_i$ and $\mathsf{G}' = \Sigma_{i\in I}\Lambda_i.\mathsf{G}'_i$ with $\mathsf{G}_i \xrightarrow{\Lambda} \mathsf{G}'_i$ and $\Lambda \in cap(\mathsf{G}_i)$ and $\mathsf{prt}(\Lambda) \cap \mathsf{prt}(\Lambda_i) = \emptyset$ for all $i \in I$.
Since the last rule in the derivation of $\mathsf{G} \vdash \mathbb{M}$ must be [TComm], it follows that

- $\{\Lambda_i\}_{i\in I}$ is $\mathscr{P}$-coherent for $\mathbb{M}$;
- $\mathbb{M} \xrightarrow{\Lambda_i} \mathbb{M}_i$ and $\mathsf{G}_i \vdash^{\mathscr{P}} \mathbb{M}_i$, for all $i \in I \neq \emptyset$;
- $\mathsf{prt}(\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i) = \mathsf{prt}(\mathbb{M})$.

By the coinduction hypothesis, we know that, for each $i \in I$, there exists $\mathbb{M}'_i$ such that

$$\mathbb{M}_i \xrightarrow{\Lambda} \mathbb{M}'_i \qquad \text{and} \qquad \mathsf{G}'_i \vdash \mathbb{M}'_i$$

Notice that, being the label Λ the same for all these reductions, by Lemma 5.1 there exists a unique redex

$$\mathsf{p}[\mathsf{q}!\lambda.P \dotplus P'] \parallel \mathsf{q}[\mathsf{p}?\lambda.Q \dotplus Q']$$

with contractum $\mathsf{p}[P] \parallel \mathsf{q}[Q]$ in all the $\mathbb{M}_i$, such that $\Lambda = \mathsf{p}\lambda\mathsf{q}$. On the other hand, since we know that $\mathsf{prt}(\Lambda) \cap \mathsf{prt}(\Lambda_i) = \emptyset$ for all $i \in I$, it must be the case that $\Lambda_i = \mathsf{r}_i\lambda_i\mathsf{s}_i$ and

$$\mathbb{M}'_i \equiv \mathsf{r}_i[R_i] \parallel \mathsf{s}_i[S_i] \parallel \mathbb{M}''_i$$

for some r_i, s_i, R_i, S_i, $\mathbb{M}''_i$ and for all $i \in I$. Hence, since $\mathbb{M}_i \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathbb{M}'_i$, we have that, for each $i \in I$,

$$\mathbb{M} \equiv \mathsf{r}_i[\mathsf{s}_i!\lambda_i.R_i \dotplus R'_i] \parallel \mathsf{s}_i[\mathsf{r}?\lambda_i.S_i \dotplus S'_i] \parallel \mathbb{M}''_i \xrightarrow{\mathsf{p}\lambda\mathsf{q}} \mathsf{r}_i[\mathsf{s}_i!\lambda_i.R_i \dotplus R'_i] \parallel \mathsf{s}_i[\mathsf{r}?\lambda_i.S_i \dotplus S'_i] \parallel \mathbb{M}'''_i \equiv \mathbb{M}'$$

for some R'_i, S'_i (if any), $\mathbb{M}'''_i$ and for all $i \in I$.

By Lemma 5.3, $\mathsf{G}'_i \vdash \mathsf{r}_i[R_i] \parallel \mathsf{s}_i[S_i] \parallel \mathbb{M}''_i$ implies $\mathsf{prt}(\mathsf{G}'_i) = \mathsf{prt}(\mathsf{r}_i[R_i] \parallel \mathsf{s}_i[S_i] \parallel \mathbb{M}''_i)$ and then $\mathsf{prt}(\mathsf{G}') = \mathsf{prt}(\mathbb{M}')$. Moreover, from $\mathsf{prt}(\Lambda) \cap \mathsf{prt}(\Lambda_i) = \emptyset$ for all $i \in I$, we immediately get that $\Lambda \notin \{\Lambda_i\}_{i\in I}$. So, by Lemma 5.6 we get that $\{\Lambda_i\}_{i\in I}$ is $\mathscr{P}$-coherent for $\mathbb{M}'$. We conclude that there exists a derivation ending by the following application of Rule [TComm]

$$[\text{TComm}]\ \frac{\mathbb{M}' \xrightarrow{\Lambda_i} \mathbb{M}'_i \quad \mathsf{G}'_i \vdash \mathbb{M}'_i \quad \forall i \in I \neq \emptyset \qquad \{\Lambda_i\}_{i\in I} \text{ is } \mathscr{P}\text{-coherent for } \mathbb{M}' \quad \mathsf{prt}(\mathsf{G}') = \mathsf{prt}(\mathbb{M}')}{\mathsf{G}' \vdash^{\mathscr{P}} \mathbb{M}'}$$

□

Toward establishing the property that typable sessions are lock free, we first prove the following lemma. In words, if $\mathsf{p} \in \mathsf{prt}(\mathsf{G})$, then it must occur somewhere in its syntactic tree, hence there is a trace $\sigma \cdot \Lambda$ out of G, consisting just of external communications, which corresponds to a path in the tree ending by the first communication label Λ involving p.

Lemma 5.9 *If $\mathsf{p} \in \mathsf{prt}(\mathsf{G})$, then there are σ, Λ and G' such that $\mathsf{G} \xrightarrow{\sigma \cdot \Lambda} \mathsf{G}'$, $\mathsf{p} \notin \mathsf{prt}(\sigma)$ and $\mathsf{p} \in \mathsf{prt}(\Lambda)$.*

Proof. The proof is by coinduction on G. Since $\mathsf{p} \in \mathsf{prt}(\mathsf{G})$ we have that $\mathsf{G} = \Sigma_{i \in I} \Lambda_i.\mathsf{G}_i$. Now, let us assume $\mathsf{p} \in \bigcup_{i \in I} \mathsf{prt}(\Lambda_i)$. Without loss of generality, we can also assume that $\mathsf{p} \in \mathsf{prt}(\Lambda_j)$ for some $j \in I$. Then we immediately have that $\mathsf{G} \xrightarrow{\Lambda_j} \mathsf{G}_j$ by Axiom [E-Comm], and the thesis trivially follows by taking $\sigma = \varepsilon$. Otherwise, since $\mathsf{p} \in \mathsf{prt}(\mathsf{G}) = \bigcup_{i \in I} \mathsf{prt}(\Lambda_i) \cup \bigcup_{i \in I} \mathsf{prt}(\mathsf{G}_i)$, we have that $\mathsf{p} \notin \bigcup_{i \in I} \mathsf{prt}(\Lambda_i)$ implies $\mathsf{p} \in \mathsf{prt}(\mathsf{G}_j)$ for some $j \in I$. By the coinduction hypothesis, we have that there are a σ' and a Λ such that $\mathsf{G}_j \xrightarrow{\sigma' \cdot \Lambda} \mathsf{G}'$, $\mathsf{p} \notin \mathsf{prt}(\sigma')$ and $\mathsf{p} \in \mathsf{prt}(\Lambda)$. Then the thesis follows by setting $\sigma = \Lambda_j \cdot \sigma'$, since $\mathsf{G} \xrightarrow{\Lambda_j} \mathsf{G}_j$ by Axiom [E-Comm] and $\mathsf{G}_j \xrightarrow{\sigma' \cdot \Lambda} \mathsf{G}'$. □

Observe that the last lemma is a sort of inverse implication w.r.t. Lemma 5.4, since it shows that the existence of a capability which is an actual communication of a global type G follows by the fact that one of the involved participants is in $\mathsf{prt}(\mathsf{G})$.

We are now in place to prove that typable sessions are lock free.

Theorem 5.10 (Lock Freedom) *If $\mathbb{M}$ is typable, then $\mathbb{M}$ is lock free.*

Proof. Let $\mathsf{G} \vdash^{\mathscr{P}} \mathbb{M}$. Following Definition 2.5, in order to prove Lock Freedom for $\mathbb{M}$, let $\mathbb{M} \xrightarrow{\sigma} \mathbb{M}'$ for a finite σ and let $\mathsf{p} \in \mathsf{prt}(\mathbb{M}')$. By Subject Reduction (Theorem 5.7) we get $\mathsf{G}' \vdash^{\mathscr{P}} \mathbb{M}'$. We can now recur to Lemma 5.3 and get $\mathsf{p} \in \mathsf{prt}(\mathsf{G}')$. From the fact that $\mathsf{p} \in \mathsf{prt}(\mathsf{G}')$ and by Lemma 5.9 it follows that $\mathsf{G}' \xrightarrow{\sigma' \cdot \Lambda} \mathsf{G}''$ for some σ' and Λ with $\mathsf{p} \notin \mathsf{prt}(\sigma')$ and $\mathsf{p} \in \mathsf{prt}(\Lambda)$. Now the thesis follows by Session Fidelity (Theorem 5.8). □

We conclude this section by showing that typability of a session does depend on the choice neither of the $\mathscr{P}$-coherent sets nor of the partition $\mathscr{P}$.

Theorem 5.11 *If $\mathbb{M}$ is typable in $\vdash^{\mathscr{P}}$ and $\{\Lambda_i\}_{i \in I}$ is $\mathscr{P}$-coherent for $\mathbb{M}$, then there are G_i for $i \in I$ such that $\Sigma_{i \in I} \Lambda_i.\mathsf{G}_i \vdash^{\mathscr{P}} \mathbb{M}$.*

Proof. The $\mathscr{P}$-coherence of $\{\Lambda_i\}_{i \in I}$ for $\mathbb{M}$ gives $\mathbb{M} \xrightarrow{\Lambda_i} \mathbb{M}_i$, which implies by Subject Reduction (Theorem 5.7) $\mathsf{G}_i \vdash^{\mathscr{P}} \mathbb{M}_i$ for some G_i and all $i \in I$. By Lemma 5.3 $\mathsf{prt}(\mathsf{G}_i) = \mathsf{prt}(\mathbb{M}_i)$ for all $i \in I$. From $\mathbb{M} \xrightarrow{\Lambda_i} \mathbb{M}_i$ for all $i \in I$ we get $\mathsf{prt}(\mathbb{M}) = \bigcup_{i \in I} \mathsf{prt}(\Lambda_i) \cup \bigcup_{i \in I} \mathsf{prt}(\mathbb{M}_i)$ by Lemma 5.2. By definition, $\mathsf{prt}(\Sigma_{i \in I} \Lambda_i.\mathsf{G}_i) = \bigcup_{i \in I} \mathsf{prt}(\Lambda_i) \cup \bigcup_{i \in I} \mathsf{prt}(\mathsf{G}_i)$. We conclude $\mathsf{prt}(\Sigma_{i \in I} \Lambda_i.\mathsf{G}_i) = \mathsf{prt}(\mathbb{M})$, so we can derive $\Sigma_{i \in I} \Lambda_i.\mathsf{G}_i \vdash^{\mathscr{P}} \mathbb{M}$ using Rule [TComm]. □

Theorem 5.12 *If $\mathbb{M}$ is typable in $\vdash^{\mathscr{P}}$ and it is $\mathscr{P}'$-modularisable, then $\mathbb{M}$ is typable in $\vdash^{\mathscr{P}'}$ too.*

Proof. Since $\mathbb{M}$ is typable in $\vdash^{\mathscr{P}}$, then $\mathbb{M}$ is $\mathscr{P}$-modularisable by Definition 4.2. Let $\mathscr{P} = \{\mathbf{P}_k\}_{k \in K}$ and $\{\Lambda_h^k\}_{h \in H_k} = \{\Lambda \in \mathscr{L}(\mathbb{M}) \mid \mathsf{prt}(\Lambda) \cap \mathbf{P}_k \neq \emptyset\}$. By definition of partition we observe that $\bigcup_{k \in K} \bigcup_{h \in H_k} \Lambda_h^k = \mathscr{L}(\mathbb{M})$. By Definitions 4.2 and 3.4 for all $k \in K$ $\{\Lambda_h^k\}_{h \in H_k}$ is $\mathscr{P}$-coherent for $\mathbb{M}$. By Lemma 5.11 for each $k \in K$ there is a global type $\mathsf{G}_k = \Sigma_{h \in H_k} \Lambda_h^k.\widehat{\mathsf{G}}_h$ which can be assigned to $\mathbb{M}$ in $\vdash^{\mathscr{P}}$, that is $\Sigma_{h \in H_k} \Lambda_h^k.\widehat{\mathsf{G}}_h \vdash^{\mathscr{P}} \mathbb{M}$.
Let $\{\Lambda_i\}_{i \in I}$ be $\mathscr{P}'$-coherent for $\mathbb{M}$. We proceed now by coinduction simultaneously on the derivations $\Sigma_{h \in H_k} \Lambda_h^k.\widehat{\mathsf{G}}_h \vdash^{\mathscr{P}} \mathbb{M}$ for all $k \in K$. From the above, for each $i \in I$ there is $k_i \in K$ and $l_i \in H_{k_i}$ such that one

of the premises of the conclusion $\Sigma_{h\in H_{k_i}}\Lambda_h^{k_i}.\widehat{\mathsf{G}}_h \vdash^{\mathscr{P}} \mathbb{M}$ is $\widehat{\mathsf{G}}_{l_i} \vdash^{\mathscr{P}} \mathbb{M}_i$ where $\mathbb{M} \xrightarrow{\Lambda_i} \mathbb{M}_i$. By coinduction we get $\mathsf{G}_i'' \vdash^{\mathscr{P}'} \mathbb{M}_i$ for some G_i'' and for all $i \in I$. By Lemma 5.3 $\mathsf{prt}(\mathsf{G}_i'') = \mathsf{prt}(\mathbb{M}_i)$ for all $i \in I$. We have $\mathsf{prt}(\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i'') = \bigcup_{i\in I}\mathsf{prt}(\Lambda_i) \cup \bigcup_{i\in I}\mathsf{prt}(\mathsf{G}_i'')$ and $\mathsf{prt}(\mathbb{M}) = \bigcup_{i\in I}\mathsf{prt}(\Lambda_i) \cup \bigcup_{i\in I}\mathsf{prt}(\mathbb{M}_i)$ by Lemma 5.2, so we get $\mathsf{prt}(\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i'') = \mathsf{prt}(\mathbb{M})$. Then we can use Rule [TCOMM] to derive $\Sigma_{i\in I}\Lambda_i.\mathsf{G}_i'' \vdash^{\mathscr{P}'} \mathbb{M}$. □

6 Concluding Remarks, Related and Future Works

In the setting of the *message-passing* communication model, it is possible to envisage mechanisms of interaction where a process can be, at the very same time, both a potential sender and a potential receiver. In various frameworks for concurrent systems, like session types and communicating finite state machines, as well as the π-calculus, such mechanism is referred to as *mixed choice*. The flexibility and expressive power of mixed choice is understandably counterbalanced by a difficult control of the behaviour of systems. That was arguably the motivation that mostly restrained the session type community from pursuing a thorough investigation of this sort of interactions. A stimulus in that direction has been instead recently given by some papers like [10, 28, 29, 30, 31]. In particular, [10, 30] investigate mixed choice for binary session types, whereas [31] considers mixed choice in a MPST setting [18, 19] following the approach of [32] (global types are in fact not taken into account in [31]). Even if the main concern of [31] is the expressivity of multiparty calculi (according to the full range of possible restrictions of mixed choice), type systems assigning local types to processes are provided, where various predicates on contexts of local types are investigated. In [28, 29] binary sessions with timeout and mixed choice are enriched with a semantics guaranteeing Progress and a type system enjoying Subject Reduction.

Inspired by [31], we carry on an investigation on the use of mixed choice for synchronous communications in the setting of SMPS [12, 4]. In SMPS, global types are inferred for sessions, i.e. parallel compositions of named processes, the latter being an abstraction for both processes and local types usually considered in MPST. Our processes can use now mixed choice. Subject Reduction, Session Fidelity, as well as Lock Freedom are ensured for typable sessions. The most relevant aspect of our type system is that we look at sessions as implicitly composed by modules whose participants freely interact via unrestricted mixed choice, whereas the inter-module communications can be more easily controlled by allowing communications with only one participant. Such an approach does not discard a priori any session. In fact all sessions – even those developed without any specific modularisation in mind – can be modularised in a less or more refined way: from a single large module comprising the whole session, to a set of modules made by single participants. In the former case the typing is less effective, since the global type would result in a complete interaction tree, whereas in the latter case the typing coincides with the standard SMPS (with no mixed choice). In particular, our type system is conservative since, for processes without mixed choice, it coincides with the type system of [6], which is at present one of the most expressive. For mixed choice there is only the type system of [31], which is modularised by predicates on local types. An interesting property ensured by that type system is *safety*. Safety in [31] entails that the protocol for process interactions is such that, when a participant intends to perform an interaction, it nondeterministically chooses among all the participants that can interact with it. Then there is a nondeterministic decision concerning who has to play the role of the sender and who of the receiver. At that point, the sender performs an internal choice among the available outputs. Typing then guarantees that the possibility of interaction does not depend on the chosen output. We consider, instead, a simplified synchronisation protocol where there exists a nondeterministic choice among all the participants that can actually interact and all the possible communication interactions. As mentioned in Section 2, this approach makes "!" and "?" just two complementary synchronisation actions. It is however possible to

modify our type system in order to guarantee safety of sessions as defined in [31] by requiring that

$$\mathsf{p}[\mathsf{q}!\lambda.P \dotplus P'] \in \mathbb{M} \text{ and } \mathsf{q}[\mathsf{p}?\lambda'.Q \dotplus Q'] \in \mathbb{M} \text{ imply } \mathsf{p}\lambda\mathsf{q} \in \mathscr{L}(\mathbb{M})$$

The safety condition only ensures that an output finds the corresponding input if the receiver offers some inputs for the sender. An alternative condition is

$$\mathsf{p}[\mathsf{q}!\lambda.P \dotplus P'] \in \mathbb{M} \text{ implies } \mathbb{M} \xrightarrow{\sigma\cdot\mathsf{p}\lambda\mathsf{q}} \text{ for some } \sigma$$

With this last condition we would type less sessions, for example we would not type

$$\mathsf{p}[\mathsf{q}!\lambda + \mathsf{r}!\lambda'] \parallel \mathsf{r}[\mathsf{p}?\lambda']$$

and the election example. The feature of this alternative condition is that the choice between outputs is internal, in agreement with an asynchronous implementation. In such a case it is worth remarking that a restriction of the subtyping relation used in [31] would be implicitly entailed by our typing.

As first pointed out in [14], the naive extension of the original type system [19] to sessions where input choices have different senders is unsound. In fact one can type sessions which reduce to untypable and stuck sessions. Suitable conditions ensuring a sound extension have been proposed both for the synchronous [14] and asynchronous [24, 11] communications. Notably, the type system proposed here does not have this problem.

We notice that modular sessions can be obtained by connecting independent sessions via gateways, according to the PaI approach to system composition. When composing several typable SMPS systems (through *compatible* interfaces) one gets a typable system [7]. Although the presence of mixed choice does not seem to be a major obstacle, it is unlikely a result like the one of [7] could be easily translated in the present context. In fact, as shown in Example 3.5 we allow the presence of multiple connectors per module. This possibility is actually a severe impediment to the safeness of PaI composition. In fact, let us take the following two typable sessions:

$$\mathsf{p}[\mathsf{q}!\lambda] \parallel \mathsf{q}[\mathsf{p}?\lambda] \qquad\qquad \mathsf{r}[\mathsf{s}?\lambda] \parallel \mathsf{s}[\mathsf{r}!\lambda]$$

Taking all the participants as interfaces and considering that there exists a typable connection policy among them [7], the PaI composition would result in the following untypable $\{\{\mathsf{p},\mathsf{q}\},\{\mathsf{r},\mathsf{s}\}\}$-modular session

$$\mathsf{p}[\mathsf{r}?\lambda.\mathsf{q}!\lambda] \parallel \mathsf{q}[\mathsf{p}?\lambda.\mathsf{s}!\lambda] \parallel \mathsf{r}[\mathsf{s}?\lambda.\mathsf{p}!\lambda] \parallel \mathsf{s}[\mathsf{q}?\lambda.\mathsf{r}!\lambda]$$

where all the processes begin with an input, so forming a deadlock. This problem was overcome in [13], in a setting using projections and without mixed choice, by means of a suitable extension of the syntax of global types.

It is worth noticing that mixed choice make PaI composition problematic even by allowing one connector only. For example, by composing using p and t the following typable sessions

$$\mathsf{p}[\mathsf{q}!\lambda_1 + \mathsf{r}?\lambda_2] \parallel \mathsf{q}[\mathsf{p}?\lambda_1 + \mathsf{s}?\lambda_3] \parallel \mathsf{r}[\mathsf{p}!\lambda_2 + \mathsf{s}!\lambda_4] \parallel \mathsf{s}[\mathsf{q}!\lambda_3 + \mathsf{r}?\lambda_4]$$
$$\mathsf{t}[\mathsf{u}?\lambda_1 + \mathsf{v}!\lambda_2] \parallel \mathsf{u}[\mathsf{t}!\lambda_1 + \mathsf{w}!\lambda_3] \parallel \mathsf{v}[\mathsf{t}?\lambda_2 + \mathsf{w}?\lambda_4] \parallel \mathsf{w}[\mathsf{u}?\lambda_3 + \mathsf{v}!\lambda_4]$$

we get the session

$$\mathsf{p}[\mathsf{t}?\lambda_1.\mathsf{q}!\lambda_1 + \mathsf{r}?\lambda_2.\mathsf{t}!\lambda_2] \parallel \mathsf{q}[\mathsf{p}?\lambda_1 + \mathsf{s}?\lambda_3] \parallel \mathsf{r}[\mathsf{p}!\lambda_2 + \mathsf{s}!\lambda_4] \parallel \mathsf{s}[\mathsf{q}!\lambda_3 + \mathsf{r}?\lambda_4] \parallel$$
$$\mathsf{t}[\mathsf{u}?\lambda_1.\mathsf{p}!\lambda_1 + \mathsf{p}?\lambda_2.\mathsf{v}!\lambda_2] \parallel \mathsf{u}[\mathsf{t}!\lambda_1 + \mathsf{w}!\lambda_3] \parallel \mathsf{v}[\mathsf{t}?\lambda_2 + \mathsf{w}?\lambda_4] \parallel \mathsf{w}[\mathsf{u}?\lambda_3 + \mathsf{v}!\lambda_4]$$

which reduces to the stuck session

$$\mathsf{p}[\mathsf{t}!\lambda_2] \parallel \mathsf{t}[\mathsf{p}!\lambda_1]$$

PaI composition provides also an intuitive justification for the shape of connectors in our modularisation. In fact, by composing systems via interfaces with unrestricted mixed choice, most of the communication properties, if any, are not preserved, as shown by the previous example.

As future work we plan to investigate PaI composition for sessions with mixed choice and asyn-

chronous communication, taking inspiration from [28, 29], where asynchronous communication for sessions with mixed choice is first modelled. We deem worth investigating the modular approach to session types also for standard MPST, as well as for the recent approaches to global types and projections devised in [24, 23].

Acknowledgments We wish to gratefully thank the anonymous reviewers for their thoughtful and helpful comments.

References

[1] Franco Barbanera, Viviana Bono & Mariangiola Dezani-Ciancaglini (2025): *Open compliance in multiparty sessions with partial typing*. Journal of Logical and Algebraic Methods in Programming 144, p. 101046, doi:10.1016/j.jlamp.2025.101046.

[2] Franco Barbanera, Viviana Bono & Mariangiola Dezani-Ciancaglini (2025): *Partially typed multiparty sessions with internal delegation*. Journal of Logical and Algebraic Methods in Programming 142, p. 101018, doi:10.1016/J.JLAMP.2024.101018.

[3] Franco Barbanera & Mariangiola Dezani-Ciancaglini (2023): *Partially typed multiparty sessions*. In Clément Aubert, Cinzia Di Giusto, Simon Fowler & Larisa Safina, editors: *ICE*, *EPTCS* 383, Open Publishing Association, pp. 15–34, doi:10.4204/EPTCS.383.2.

[4] Franco Barbanera, Mariangiola Dezani-Ciancaglini & Ugo de'Liguoro (2022): *Open compliance in multiparty sessions*. In S. Lizeth Tapia Tarifa & José Proença, editors: *FACS*, *LNCS* 13712, Springer, pp. 222–243, doi:10.1007/978-3-031-20872-0_13.

[5] Franco Barbanera, Mariangiola Dezani-Ciancaglini & Ugo de'Liguoro (2024): *Partial typing for asynchronous multiparty sessions*. In Sandra Alves & Ian Mackie, editors: *DCM*, *EPTCS* 408, Open Publishing Association, pp. 1–20, doi:10.4204/EPTCS.408.1.

[6] Franco Barbanera, Mariangiola Dezani-Ciancaglini & Ugo de'Liguoro (2024): *Un-projectable global types for multiparty sessions*. In Alessandro Bruni, Alberto Momigliano, Matteo Pradella & Matteo Rossi, editors: *PPDP*, ACM Press, pp. 15:1–15:13, doi:10.1145/3678232.3678245.

[7] Franco Barbanera, Mariangiola Dezani-Ciancaglini, Lorenzo Gheri & Nobuko Yoshida (2023): *Multicompatibility for multiparty-session composition*. In Santiago Escobar & Vasco T. Vasconcelos, editors: *PPDP*, ACM Press, pp. 2:1–2:15, doi:10.1145/3610612.3610614.

[8] Luc Bougé (1988): *On the existence of symmetric algorithms to find leaders in networks of communicating sequential processes*. Acta Informaticae 25(2), p. 179–201, doi:10.1007/BF00263584.

[9] Daniel Brand & Pitro Zafiropulo (1983): *On communicating finite-state machines*. Journal of ACM 30(2), pp. 323–342, doi:10.1145/322374.322380.

[10] Filipe Casal, Andreia Mordido & Vasco T. Vasconcelos (2022): *Mixed sessions*. Theoretical Computer Science 897, pp. 23–48, doi:10.1016/J.TCS.2021.08.005.

[11] Ilaria Castellani, Mariangiola Dezani-Ciancaglini & Paola Giannini (2022): *Asynchronous Sessions with Input Races*. In Marco Carbone & Rumyana Neykova, editors: *PLACES*, *EPTCS* 356, Open Publishing Association, pp. 12–23, doi:10.4204/EPTCS.356.2.

[12] Francesco Dagnino, Paola Giannini & Mariangiola Dezani-Ciancaglini (2023): *Deconfined global types for asynchronous sessions*. Logical Methods in Computer Science 19(1), pp. 1–41, doi:10.46298/lmcs-19(1:3)2023.

[13] Lorenzo Gheri & Nobuko Yoshida (2023): *Hybrid multiparty session types: compositionality for protocol specification through endpoint projection*. PACMPL 7 (OOPSLA1), pp. 112–142, doi:10.1145/3586031.

[14] Rob van Glabbeek, Peter Höfner & Ross Horne (2021): *Assuming just enough fairness to make session types complete for lock-freedom*. In Leonid Libkin, editor: *LICS*, IEEE, pp. 1–13, doi:10.1109/LICS52264.2021.9470531.

[15] Mohamed G. Gouda, Eric G. Manning & Yao-Tin Yu (1984): *On the progress of communication between two machines*. Information and Control 63(3), pp. 200–2016, doi:10.1016/S0019-9958(84)80014-5.

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

[17] Kohei Honda, Vasco T. Vasconcelos & Makoto Kubo (1998): *Language primitives and type discipline for structured communication-based programming*. In Chris Hankin, editor: *ESOP*, *LNCS* 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.

[18] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In George C. Necula & Philip Wadler, editors: *POPL*, ACM Press, pp. 273–284, doi:10.1145/1328897.1328472.

[19] Kohei Honda, Nobuko Yoshida & Marco Carbone (2016): *Multiparty asynchronous session types*. Journal of the ACM 63(1), pp. 9:1–9:67, doi:10.1145/2827695.

[20] Naoki Kobayashi & Davide Sangiorgi (2010): *A hybrid type system for lock-freedom of mobile processes*. ACM Transactions on Programming Languages and Systems 32(5), pp. 16:1–16:49, doi:10.1016/j.jlamp.2024.101018.

[21] Dexter Kozen & Alexandra Silva (2017): *Practical coinduction*. Mathematical Structures in Computer Science 27(7), pp. 1132–1152, doi:10.1017/S0960129515000493.

[22] Julien Lange, Nicholas Ng, Bernardo Toninho & Nobuko Yoshida (2017): *Fencing off go: liveness and safety for channel-based programming*. In Giuseppe Castagna & Andrew D. Gordon, editors: *POPL*, ACM Press, pp. 748–761, doi:10.1145/3009837.3009847.

[23] Elaine Li, Felix Stutz, Thomas Wies & Damien Zufferey (2023): *Complete multiparty session type projection with automata*. In Constantin Enea & Akash Lal, editors: *CAV*, *LNCS* 13966, Springer, pp. 350–373, doi:10.1007/978-3-031-37709-9_17.

[24] Rupak Majumdar, Madhavan Mukund, Felix Stutz & Damien Zufferey (2021): *Generalising projection in asynchronous multiparty session types*. In Serge Haddad & Daniele Varacca, editors: *CONCUR*, *LIPIcs* 203, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 35:1–35:24, doi:10.4230/LIPIcs.CONCUR.2021.35.

[25] https://www.ottia.com/en/post/project-modularization.

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

[27] Catuscia Palamidessi (2003): *Comparing the expressive power of the synchronous and asynchronous pi-calculi*. Mathematical Structures in Computer Science 13(5), pp. 685–719, doi:10.1017/S0960129503004043.

[28] Jonah Pears, Laura Bocchi & Andy King (2023): *Safe Asynchronous Mixed-Choice for Timed Interactions*. In Sung-Shik Jongmans & Antónia Lopes, editors: *COORDINATION*, *LNCS* 13908, Springer, pp. 214–231, doi:10.1007/978-3-031-35361-1_12.

[29] Jonah Pears, Laura Bocchi, Maurizio Murgia & Andy King (2024): *Introducing TOAST: Safe Asynchronous Mixed-Choice For Timed Interactions*. CoRR abs/2401.11197, doi:10.48550/ARXIV.2401.11197.

[30] Kirstin Peters & Nobuko Yoshida (2024): *Mixed choice in session types*. Information and Computation 298, p. 105164, doi:10.1016/J.IC.2024.105164.

[31] Kirstin Peters & Nobuko Yoshida (2024): *Separation and encodability in mixed choice multiparty sessions*. In Pawel Sobocinski, Ugo Dal Lago & Javier Esparza, editors: *LICS*, ACM Press, pp. 62:1–62:15, doi:10.1145/3661814.3662085.

[32] Alceste Scalas & Nobuko Yoshida (2019): *Less is more: multiparty session types revisited*. PACMPL 3 (POPL), pp. 30:1–30:29, doi:10.1145/3290343.

[33] Herbert A. Simon (1991): *The architecture of complexity*. In: Facets of Systems Science, Springer, pp. 457–476, doi:10.1007/978-1-4899-0718-9_31.

Reactive Semantics for User Interface Description Languages

Basile Pesin
basile.pesin@enac.fr

Celia Picard
celia.picard@enac.fr

Cyril Allignol
cyril.allignol@enac.fr

Fédération ENAC ISAE-SUPAERO ONERA, Université de Toulouse, France

User Interface Description Languages (UIDLs) are high-level languages that facilitate the development of Human-Machine Interfaces, such as Graphical User Interface (GUI) applications. They usually provide first-class primitives to specify how the program reacts to an external event (user input, network message), and how data flows through the program. Although these domain-specific languages are now widely used to implement safety-critical GUIs, little work has been invested in their formalization and verification.

In this paper, we propose a denotational semantic model for a core reactive UIDL, Smalite, which we argue is expressive enough to encode constructs from more realistic languages. This preliminary work may be used as a stepping stone to produce a formally verified compiler for UIDLs.

1 Introduction and Context

With the democratization of interactive devices, the general interaction paradigm has changed. Users now expect to interact with their systems through tactile interactions or advanced interactive devices. This extends to critical systems too, notably aviation related ones. For instance, aircraft cockpits are now digital and tactile [13] and paper strips for air traffic controllers have been replaced by elaborate digital dashboards [10].

Using traditional programming languages to implement the interactive parts of systems implies the use of numerous callbacks. The resulting code quickly becomes spaghetti code and gets particularly difficult to analyze and maintain [16, 17]. The use of dedicated languages has shown to be beneficial in such circumstances [18]. Such languages are called User Interface Description Languages (UIDLs). These languages can efficiently describe both the appearance of the system (its scene graph) and the interactive behavior (mainly activation propagation). They are becoming more popular, including for the development of critical systems [2, 1]. However, these languages have not been formally defined, particularly not their semantics, which hinders any formal reasoning on these safety-critical programs. In this paper, we tackle this problem by proposing denotational semantics for a minimal declarative UIDL.

There has been a few other attempts to give a formal semantics to interactive languages or libraries such as React [14]. However, React is based on a functional language and as such, has very different mechanisms than other UIDL/interactive languages. Indeed, the behavior of a React program is described using small-step operational semantics which specify the order in which components are rendered. In this paper, we focus on a declarative language with denotational semantics where the order of updates is left implicit.

Previous work has paved the way towards a minimal common abstract syntax for Smala, a declarative UIDL [15]. A first formal semantic model based on bigraphs has been proposed [20], but in this work we explore an alternative expression of the semantics, which is simpler and should facilitate compiler

C. Aubert, C. Di Giusto, S. Fowler & V. K. I Pun (Eds.):
18th Interaction and Concurrency Experience (ICE 2025)
EPTCS 425, 2025, pp. 21–35, doi:10.4204/EPTCS.425.3

verification in the fashion of CompCert [12] or Vélus [4]. To give more confidence in those semantics, we also aim at proving the equivalence between the two semantics, as future work.

Hence, we propose a new denotational semantics for Smalite, a declarative UIDL which includes a minimal set of constructs from Smala. As a preliminary step to a compiler correctness proof, we mechanized the language and its semantics in the Rocq prover [1] [23], and implemented a prototype compiler that generates imperative code. In section 2, we give an informal presentation of our language through an example program implementing a simple Graphical User Interface (GUI) program. Then, in section 3, we describe our formalization of the reactive semantics of the language. Finally, in section 4, we discuss the future steps towards extending the language and building a formal proof of correctness for its compiler.

2 Specifying interactions with Smala

The language we propose closely resembles Smala [15], a UIDL used in safety-critical applications [2, 7]. In fig. 1, we present an example Smala program where two buttons control a counter by either decrementing it until it reaches 0 or setting it back to 3.

Smala programs are composed of named *processes*. The root process is a `Component` (`line 1`), usually named `root`, which contains child processes. The counter is implemented as a *property* `count` (`line 2`), which is declared with a type and an expression giving its initial value. Then, the program contains a `Spike` (`line 3`), which represents an event that may be triggered from inside or outside the program, and reacted to. This particular `Spike` represents the event "the counter has just reached 0". Indeed, it is triggered by the *binding* on the same line, whose left-hand side is a condition checking that `count` equals 0. Conditions are checked only at reactions where the value of one of the properties involved changes; therefore, `zero` is triggered only at reactions where the value of `count` changes to 0.

The next process is a `Frame`, which is a special *graphical* component (`line 6`). For the purpose of our reactive semantics, a `Frame` is just syntactic sugar for a component with properties `title`, `width` and `height`, passed as parameters between parentheses, as well as a `Spike` `close`. On the other hand, our prototype compiler generates code which uses the `Frame` and its parameters to open a window. The compiler also binds user requests to close the window (e.g. by clicks on the ✕ button) to triggering the `close` `Spike`. Therefore, the program can define custom reactions to user actions.

The first child of the `Frame` defines the `Font` used to display text in the interface (`line 7`). Since the rest of the program does not need to refer to this particular component, its name is unspecified (`_`). Then, the frame contains three main components: two buttons, and a text label. Each button is implemented by a component `FillColor` (with parameters `red`, `green` and `blue`), which sets the color of its `Rectangle` child (with parameters `x`, `y`, `width` and `height`). Text labels are implemented by a `FillColor` which sets the color of its `Text` child (with parameters `text`, `x` and `y`).

`Rectangle` components have two predefined spikes, `press` and `release`, which are respectively triggered when the mouse is pressed/released while the mouse pointer is on top of the rectangle. The logic of each button is implemented as a reaction to these spikes: on `press`, the button is highlighted by increasing the `green` property of its `FillColor` (`lines 14, 26`). On `release`, the `green` component is set back to its default value (`lines 15, 27`). The graphical effect of these first two reactions are shown at right of fig. 1.

The `release` `Spike` of each button is bound to an action on the counter. Releasing `decr` decreases the counter (`line 17`) by setting its new value to its previous value (accessed with `last`) minus one.

[1] formerly known as Coq

```
1  Component root {
2    Int count 3;
3
4    Spike zero; (count == 0) -> zero;
5
6    Frame f ("ICE␣2025", 300, 50) {
7      Font _ ("arial.ttf", 20) {
8        FillColor btn1 (150,150,150) {
9          Rectangle r (0,0,100,f.height) {
10           FillColor _ (0,0,0) {
11             Text _ ("decr", r.x + 30, 13) {}
12           };
13         };
14         r.press -> hg; hg: 255 =: green;
15         r.release -> dhg; dhg: 150 =: green;
16       };
17       btn1.r.release -> dec; dec: last count-1 =: count;
18       zero ->! btn1.r; (count > 0) -> btn1.r;
19
20       FillColor btn2 (150,150,150) {
21         Rectangle<d> r (btn1.r.width+10,0,100,f.height) {
22           FillColor _ (0,0,0) {
23             Text t ("restart", r.x + 20, 13) {}
24           };
25         };
26         r.press -> hg; hg: 255 =: green;
27         r.release -> dhg; dhg: 150 =: green;
28       };
29       btn2.r.release -> rst; rst: 3 =: count;
30       (count < 3) -> btn2.r; (count == 3) ->! btn2.r;
31
32       FillColor _ (255,255,255) {
33         Text t ("rem:␣" + str(count), btn2.r.x+btn2.r.width+10, 13) {
34           count -> chtext; chtext: "rem:␣" + str(count) =: text;
35         };
36       }
37     }
38   };
39
40   Exit e (0) { f.close -> trigger };
41 }
```

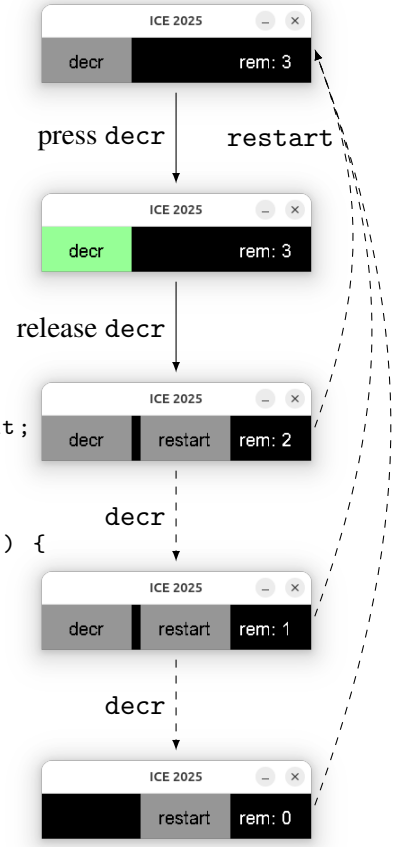


Figure 1: An example Smala program

Releasing `restart` sets the counter back to 3 (`line 29`). Buttons may be activated or deactivated depending on the value of the counter. If the counter equals 0, it is not possible anymore to decrease it, and therefore the `decr` button is deactivated (first binding `->!` `btn1.r` on `line 18`). As soon as the counter goes back above 0, it is reactivated. Conversely, the `restart` button is deactivated when the counter equals 3, and reactivated otherwise (`line 30`). Normally, the activation of a parent automatically activates all of its children. However, since the counter starts at 3, the `restart` button must be initially deactivated, which is specified by `<d>` on `line 21`.

The last child component of the `Frame` is a `Text` label displaying the value of `count`. It is updated every time the value of `count` changes (binding `count ->` on `line 34`).

$$
\begin{array}{ll}
p ::= ty\ x\ e & path ::= x \mid x\texttt{.}path \\
\mid \texttt{Spike}\ x & e ::= c \mid path \mid \texttt{last}\ path \mid \diamond e \mid e \oplus e \\
\mid x\texttt{: } e\ \texttt{=:}\ path & ia ::= \texttt{a} \mid \texttt{d} \\
\mid x\texttt{: } lhs\ \texttt{-><}ia\texttt{>}\ rhs & lhs ::= \texttt{T?(}path\texttt{)} \mid \texttt{A?(}path\texttt{)} \mid \texttt{D?(}path\texttt{)} \mid \texttt{C?(}path\texttt{)} \mid \texttt{(}e\texttt{)?} \\
\mid \texttt{Component<}ia\texttt{>}\ x\ \{p^*\} & rhs ::= \texttt{T!(}path\texttt{)} \mid \texttt{A!(}path\texttt{)} \mid \texttt{D!(}path\texttt{)}
\end{array}
$$

Figure 2: Abstract Syntax of Smalite

Concrete LHS	Concrete RHS	Abstract LHS	Abstract RHS	Event
`path ->`	`-> path`	`T?(`*path*`)`	`T!(`*path*`)`	`Trigger` *path*
`path ->`	N/A	`C?(`*path*`)`	N/A	`Assign` *v path*
`(e) ->`	N/A	`(`*e*`)?`	N/A	`Assign` *v path*, *path* $\in$ free(*e*)
`path ->`	`-> path`	`A?(`*path*`)`	`A!(`*path*`)`	`Activate` *path*
`path !->`	`->! path`	`D?(`*path*`)`	`D!(`*path*`)`	`Deactivate` *path*

Figure 3: Correspondence between concrete syntax, abstract syntax and events

Finally, the program contains an `Exit` component, which takes an exit code as a parameter, and exposes a `Spike` `trigger` which halts the program. The `close` `Spike` of the `Frame` is bound to `trigger`, which allows the user to close the program by closing the window.

3 Formalizing Smalite's semantics

In this paper, we define the formal semantics of Smalite, a more restricted form of Smala. Smalite is a reactive language: the execution of a program can be seen as a series of reactions to external events. We saw in the example of fig. 1 that the main type of external event is the triggering of a `Spike`, such as `close (line 40)` or `release (line 15)`. Another possible external event not showcased in the example would be an outside modification of a property. For instance, an adaptive GUI program could listen for the resizing of a `Frame` which would be encoded as a change of its `width` or `height` property.

A reaction may then update the state of the process by *activating* or *deactivating* permanent processes, or *assigning* values to properties. A reaction may also *trigger* spikes which, as seen in the example, may have an effect on the view. Therefore, we generalize external events to also account for events triggered by the program during a reaction, as defined below.

$$ev ::= \texttt{Trigger}\ path \mid \texttt{Assign}\ v\ path \mid \texttt{Activate}\ path \mid \texttt{Deactivate}\ path$$

First, an event can be the triggering of a "transient" process, which does not retain its activation between cycles (`Spike` or assign). Second, it can be an assignment to a property. Last, it can be the activation or deactivation of a "permanent" process, which retains its activation between cycles (`Component` or binding).

3.1 Abstract syntax of Smalite

We now detail the abstract syntax of Smalite as we formalize it. A minimalized version of Smala has been proposed in previous work [15]. It includes a small set of core elements that constitutes the heart

of the language, expressive enough to describe the full set of components of the whole language while minimizing the number of constructs to actually formally define and verify. The translation from Smala to this minimal set is done through a transpilation pass not described here. Our syntax is presented in fig. 2, and builds upon the one proposed previously [15].

A program is a process p. There are five different processes detailed hereafter. A process may be the declaration of a property with its type *ty*, name x and initial value e. There are two kinds of transient processes, which are triggered during a single cycle: spikes and assignments. The latter assigns the result of the evaluation of an expression e to a property specified by an absolute *path*. An expression is either a constant c, the *path* to access the value or the `last` value of a property, or a unary ($\diamond$) or binary ($\oplus$) operation. Types, constants and operations are those of the host language that the compiler targets (in the case of our prototype compiler, CompCert C).

Finally, there are two kinds of permanent processes, which retain their activation between cycles. Both kinds may be initially activated or deactivated following the *ia* flag. The first permanent process is the binding, denoted by an arrow `->`. It binds the event specified by its left-hand-side to the event specified by its right-hand-side. An event detected by a left-hand-side may be the triggering (`T?`) of a transient process (`Spike`, assign), the activation (`A?`) or deactivation (`D?`) of a permanent process (binding, `Component`), the change of value of a property (`C?`), or a condition being true after one of its free variable changes (`(`e`)?`). Right-hand-sides may trigger a transient process (`T!`) or activate (`A!`) or deactivate (`D!`) a permanent process. The second permanent process is the component, which contains a list of sub-processes.

This abstract syntax is sligtly different from the concrete syntax used for the example. First, each process is explicitly named. Second, there is only one generic `Component`. It can then be instantiated to create all other components, particularly the graphical components (`Frame`, `Rectangle`, etc), as those do not play a special role in the semantics. Third, as we have seen, the left- and right-hand sides of bindings distinguish between triggering of transient processes, activation/deactivation of permanent processes, and change of property value. Figure 3 recapitulates the correspondence between concrete left- and right-hand-sides, as seen in the example, and their abstract counterpart. Last, processes refer to each other using their absolute paths, which are made up of a sequence of identifiers indicating the path from the root process to the process of interest.

In practice, our prototype compiler parses the source program of fig. 1 into an Abstract Syntax Tree (AST) represented as an inductive type in Rocq. Then, a sequence of functions recursively elaborates it into a second AST representing the more restricted syntax of fig. 2, in three passes. First, the elaborator fills-in missing names by generating globally unique identifiers. Then, it expands graphical components into generic `Component` by adding the predefined properties and spikes (e.g. `title`, `width`, `frame` for a `Frame`). Finally, it type-checks the program, makes relative paths absolutes, differentiates transient and permanent processes in bindings, and builds the restricted AST.

3.2 Semantics of program initialization

We now describe the rules that specify the initialization of the system, presented in fig. 4. The first judgment, $\vdash^{\text{activ}}_{\text{init}} p_r \Downarrow A$, specifies the initial activation for a process p rooted at path r, where A is a set of paths of permanent processes. It is defined by the rules in fig. 4a. Non-permanent processes (property, spike, assignment) never appear in this set. Bindings and components appear in the set if-and-only-if they are declared with `<a>`. For a component marked with `<a>`, the sub-processes are initially activated according to the same rules.

The second judgment, $E \vdash^{\text{env}}_{\text{init}} p_r \Downarrow E'$, specifies how the initial property values are added to initial

$\boxed{\vdash^{\text{activ}}_{\text{init}} p_r \Downarrow A}$

$$\frac{}{\vdash^{\text{activ}}_{\text{init}} (ty\ x\ e)_{r_x} \Downarrow \emptyset}\ \text{IAP} \qquad \frac{}{\vdash^{\text{activ}}_{\text{init}} (\texttt{Spike}\ x)_{r_x} \Downarrow \emptyset}\ \text{IAS} \qquad \frac{}{\vdash^{\text{activ}}_{\text{init}} (x\texttt{: }e\texttt{ =: }p)_{r_x} \Downarrow \emptyset}\ \text{IAA}$$

$$\frac{}{\vdash^{\text{activ}}_{\text{init}} (x\texttt{: }lhs\texttt{ -><d> }rhs)_{r_x} \Downarrow \emptyset}\ \text{IAB}_1 \qquad \frac{}{\vdash^{\text{activ}}_{\text{init}} (x\texttt{: }lhs\texttt{ -><a> }rhs)_{r_x} \Downarrow \{r_x.x\}}\ \text{IAB}_2$$

$$\frac{}{\vdash^{\text{activ}}_{\text{init}} (\texttt{Component<d> }x\ \{ps\})_{r_x} \Downarrow \emptyset}\ \text{IAC}_1 \qquad \frac{\forall i.\ \vdash^{\text{activ}}_{\text{init}} (ps_i)_{r_x.x} \Downarrow A_i}{\vdash^{\text{activ}}_{\text{init}} (\texttt{Component<a> }x\ \{ps\})_{r_x} \Downarrow (\bigcup_i A_i) \cup \{r_x.x\}}\ \text{IAC}_2$$

(a) Initialization of activation

$\boxed{E \vdash^{\text{env}}_{\text{init}} p_r \Downarrow E'}$

$$\frac{}{E \vdash^{\text{env}}_{\text{init}} (\texttt{Spike}\ x)_{r_x} \Downarrow E}\ \text{IES} \qquad \frac{}{E \vdash^{\text{env}}_{\text{init}} (x\texttt{: }e\texttt{ =: }p)_{r_x} \Downarrow E}\ \text{IEA} \qquad \frac{}{E \vdash^{\text{env}}_{\text{init}} (x\texttt{: }lhs\texttt{ -><_> }rhs)_{r_x} \Downarrow E}\ \text{IEB}$$

$$\frac{\emptyset, E \vdash_{\text{exp}} e \Downarrow v}{E \vdash^{\text{env}}_{\text{init}} (ty\ x\ e)_{r_x} \Downarrow E[r_x.x \mapsto v]}\ \text{IEP} \qquad \frac{}{E \vdash^{\text{env}}_{\text{init}} (\texttt{Component<_> }x\ \{\varepsilon\})_{r_x} \Downarrow E}\ \text{IEC}_1$$

$$\frac{E \vdash^{\text{env}}_{\text{init}} p_{(r_x.x)} \Downarrow E' \qquad E' \vdash^{\text{env}}_{\text{init}} (\texttt{Component<_> }x\ \{ps\})_{r_x} \Downarrow E''}{E \vdash^{\text{env}}_{\text{init}} (\texttt{Component<_> }x\ \{p;ps\})_{r_x} \Downarrow E''}\ \text{IEC}_2$$

(b) Initialization of environment

$\boxed{\vdash_{\text{init}} p \Downarrow E, A}$

$$\frac{\vdash^{\text{activ}}_{\text{init}} p_\varepsilon \Downarrow A \qquad \emptyset \vdash^{\text{env}}_{\text{init}} p_\varepsilon \Downarrow E}{\vdash_{\text{init}} p \Downarrow E, A}\ \text{I}$$

(c) Program initialization

Figure 4: Initialization rules

environment E to form E', where environments are represented by finite maps from property paths to values. It is defined by the rules in fig. 4b. As expected, spikes, assignments and bindings have no effect on the initial environment. For properties (rule IEP), the initialization expression is evaluated under the starting environment E and the resulting value is added to E. A value is either a numerical value from the C language (integer, floating-point number), a boolean value, or a string of characters. The rules for expression evaluation specifying $E, E' \vdash_{\text{exp}} e \Downarrow v$ are unsurprising, and presented in fig. 5. The notation $c^\#$ represents the semantic denotation of constant c (respectively $\diamond^\#$ for unary operator $\diamond$ and $\oplus^\#$ for binary operator $\oplus$). The current value of a variable is searched in E', the updated environment (rule EV), while its `last` value is searched in E, which represents the environment at the end of the last step, and is empty at initialization (rule EL). The evaluation of operators may fail (e.g. in case of a division by zero), which is denoted on their return value by $\lfloor v \rfloor$.

Rules IEC_1 and IEC_2, for components, highlight an interesting design choice. In our definitions, the

$$\boxed{E, E' \vdash_{\text{exp}} e \Downarrow v}$$

$$\frac{}{E, E' \vdash_{\text{exp}} c \Downarrow c^{\#}}\ \text{EC} \qquad \frac{}{E, E' \vdash_{\text{exp}} x \Downarrow E'(x)}\ \text{EV} \qquad \frac{}{E, E' \vdash_{\text{exp}} \texttt{last}\ x \Downarrow E(x)}\ \text{EL}$$

$$\frac{E, E' \vdash_{\text{exp}} e_1 \Downarrow v_1 \qquad \diamond^{\#}(v_1) = \lfloor v \rfloor}{E, E' \vdash_{\text{exp}} \diamond\, e_1 \Downarrow v}\ \text{EU} \qquad \frac{E, E' \vdash_{\text{exp}} e_1 \Downarrow v_1 \qquad E, E' \vdash_{\text{exp}} e_2 \Downarrow v_2 \qquad \oplus^{\#}(v_1, v_2) = \lfloor v \rfloor}{E, E' \vdash_{\text{exp}} e_1 \oplus e_2 \Downarrow v}\ \text{EB}$$

Figure 5: Expression evaluation rules

$$\boxed{E, A \vdash_{\text{react}} p(ev) \Downarrow E', A', T}$$

$$\frac{T = \{ev_0\} \cup \{ev \mid E, A, T, E', A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow ev\} \qquad E, T, E', A' \vdash^{\text{proc}}_{\text{safe}} p_\varepsilon \qquad E, A \vdash_{\text{update}} T \Downarrow E', A'}{E, A \vdash_{\text{react}} p(ev_0) \Downarrow E', A', (T \cap \{\texttt{Trigger}\ path \mid \texttt{Spike}\ path \in p\})}\ \text{R}$$

Figure 6: Main reaction rule

environment is updated sequentially, following the order the sub-processes are written in the program. One simpler definition would have been to update the same environment E in parallel for each sub-process ps_i into an environment E_i, and to take the union of these environments, but this would have prevented writing initial expressions that depend on the values of other properties, as seen for the button text position in fig. 1 (`line 21`). Conversely, a more expressive semantics would allow the initialization of a property to depend on the values of properties later in the program, as long as there is no cycle in the definitions. This would however require (1) a more complicated semantic judgment for initialization, (2) an analysis of the absence of cycle at the source level, and (3) a way to schedule property initialization in the compiled code. Although these are all feasible, we believe that the marginal expressivity gains are not sufficient to justify this additional complexity.

Finally, the two judgments described above are combined in rule I, presented in fig. 4c. It specifies the initialization of a Smalite program: $\vdash_{\text{init}} p \Downarrow E, A$ indicates that the process p, rooted at the empty path ε, is initialized with initial environment E and initial activation A.

3.3 Semantics of reaction

To describe how a program reacts to an external event, let us start with the main reaction rule presented in fig. 6. The judgment $E, A \vdash_{\text{react}} p(ev) \Downarrow E', A', T'$ indicates that, with initial environment E and activation A, the process p reacts to an external event ev by updating its environment to E', its activation to A', and emitting a set T of external events. The relation between these parameters are specified by three premises, which we now detail.

3.3.1 Event propagation

The first premise determines the set T of events that are emitted during the reaction step. T contains the external event that triggered the reaction, ev. It also contains any additional event propagated by the process: this is represented by judgment $E, A, T, E', A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow ev$ which can be read as "when process p

$$\boxed{E,T,E' \vdash^{\text{lhs}}_{\text{prop}} \mathit{lhs}}$$

$$\frac{\texttt{Trigger}\ x \in T}{E,T,E' \vdash^{\text{lhs}}_{\text{prop}} \texttt{T?}(x)}\ \text{PLT} \qquad \frac{\texttt{Activate}\ x \in T}{E,T,E' \vdash^{\text{lhs}}_{\text{prop}} \texttt{A?}(x)}\ \text{PLA} \qquad \frac{\texttt{Deactivate}\ x \in T}{E,T,E' \vdash^{\text{lhs}}_{\text{prop}} \texttt{D?}(x)}\ \text{PLD} \qquad \frac{\texttt{Assign}\ \text{v}\ x \in T}{E,T,E' \vdash^{\text{lhs}}_{\text{prop}} \texttt{C?}(x)}\ \text{PLC}$$

$$\frac{x \in \mathsf{free}(e) \qquad \texttt{Assign}\ \text{v}\ x \in T \qquad E,E' \vdash_{\text{exp}} e \Downarrow \texttt{true}}{E,T,E' \vdash^{\text{lhs}}_{\text{prop}} (e)\texttt{?}}\ \text{PLI}$$

$$\boxed{A,A' \vdash^{\text{rhs}}_{\text{prop}} \mathit{rhs} \Downarrow \mathit{ev}}$$

$$\frac{r_x \in A'}{A,A' \vdash^{\text{rhs}}_{\text{prop}} \texttt{T!}(r_x.x) \Downarrow \texttt{Trigger}\ r_x.x}\ \text{PRT} \qquad \frac{r_x \in A' \qquad r_x.x \notin A}{A,A' \vdash^{\text{rhs}}_{\text{prop}} \texttt{A!}(r_x.x) \Downarrow \texttt{Activate}\ r_x.x}\ \text{PRA}$$

$$\frac{r_x \in A' \qquad r_x.x \in A}{A,A' \vdash^{\text{rhs}}_{\text{prop}} \texttt{D!}(r_x.x) \Downarrow \texttt{Deactivate}\ r_x.x}\ \text{PRD}$$

$$\boxed{E,A,T,E'A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow \mathit{ev}}$$

$$\frac{p[r_x.x] = \lfloor x\texttt{:}\ \mathit{lhs}\ \texttt{->\!<\_>}\ \mathit{rhs} \rfloor \qquad r_x.x \in A' \qquad E,T,E' \vdash^{\text{lhs}}_{\text{prop}} \mathit{lhs} \qquad A,A' \vdash^{\text{rhs}}_{\text{prop}} \mathit{rhs} \Downarrow \mathit{ev}}{E,A,T,E',A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow \mathit{ev}}\ \text{PB}$$

$$\frac{p[r_x.x] = \lfloor x\texttt{:}\ e\ \texttt{=:}\ r_y.y \rfloor \qquad r_x \in A' \qquad r_y \in A' \qquad \texttt{Trigger}\ r_x.x \in T \qquad E,E' \vdash_{\text{exp}} e \Downarrow v}{E,A,T,E',A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow \texttt{Assign}\ \text{v}\ r_y.y}\ \text{PA}$$

Figure 7: Propagation rule for bindings and assignments

reacts to events T, it also emits event ev". The rules defining this judgment are presented in figs. 7 and 8. These rules are not syntax-directed: instead, each one asserts the existence of a sub-process that may propagate events. We write the judgement $p[\mathit{path}] = \lfloor p' \rfloor$ for "there is a process p' rooted at path in p".

The first set of rules, in fig. 7, describes how bindings propagate events. Rule PB, presented at the bottom, specifies that a binding only propagates events if it is activated. The third premise $E,T,E' \vdash^{\text{lhs}}_{\text{activ}} \mathit{lhs}$ of the rule assumes that the left-hand side is activated by a matching event. In particular, if the left-hand side is a condition, it must evaluate to `true`. The final premise $A,A' \vdash^{\text{rhs}}_{\text{prop}} \mathit{rhs} \Downarrow \mathit{ev}$ specifies which event ev the right-hand side emits. All rules specifying this judgment mandate that the parent of the process involved in the event is active. Activation (resp. deactivation) events are only emitted if the process was previously deactivated (resp. activated): in other words, "non-events" that do not modify the activation state are not propagated.

The last rule in fig. 7, covers the assignment. It states that an assignment produces an `Assign` v y event when (1) its parent is activated, (2) the parent of y is activated, (3) the assignment has been triggered by another event, and (4) the expression of the assignment evaluates to value v. Here, we made one interesting choice: the second premise implies that it is not possible to assign to a property whose parent is deactivated. This constraint was added because the behavior of programs where a property is assigned to when its parent is deactivated was not clear: should the assignment event be propagated during the

$$\boxed{\vdash^{\text{activ}}_{\text{prop}} p \Downarrow ev}$$

$$\frac{}{\vdash^{\text{activ}}_{\text{prop}} (x\texttt{: } lhs \texttt{ -><a> } rhs)_{r_x} \Downarrow \texttt{Activate } r_x}\ \text{PAB}$$

$$\frac{}{\vdash^{\text{activ}}_{\text{prop}} (\texttt{Component<a> } x\ \{ps\})_{r_x} \Downarrow \texttt{Activate } r_x.x}\ \text{PAC}_1 \qquad \frac{\vdash^{\text{activ}}_{\text{prop}} (ps_i)_{r_x.x} \Downarrow ev}{\vdash^{\text{activ}}_{\text{prop}} (\texttt{Component<a> } x\ \{ps\})_{r_x} \Downarrow ev}\ \text{PAC}_2$$

$$\boxed{\vdash^{\text{deactiv}}_{\text{prop}} p \Downarrow ev}$$

$$\frac{}{\vdash^{\text{deactiv}}_{\text{prop}} (x\texttt{: } lhs \texttt{ -><_> } rhs)_{r_x} \Downarrow \texttt{Deactivate } r_x.x}\ \text{PDB}$$

$$\frac{}{\vdash^{\text{deactiv}}_{\text{prop}} (\texttt{Component<_> } x\ \{ps\})_{r_x} \Downarrow \texttt{Deactivate } r_x.x}\ \text{PDC}_1 \qquad \frac{\vdash^{\text{deactiv}}_{\text{prop}} (ps_i)_{r_x.x} \Downarrow ev}{\vdash^{\text{deactiv}}_{\text{prop}} (\texttt{Component<_> } x\ \{ps\})_{r_x} \Downarrow ev}\ \text{PDC}_2$$

$$\boxed{E,A,T,E'A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow ev}$$

$$\frac{p[r_x.x] = \lfloor \texttt{Component<a> } x\ \{ps\} \rfloor \qquad \texttt{Activate } r_x.x \in T \qquad \vdash^{\text{activ}}_{\text{prop}} (ps_i)_{r_x.x} \Downarrow \texttt{Activate } y}{E,A,T,E',A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow \texttt{Activate } y}\ \text{PC}_1$$

$$\frac{p[r_x.x] = \lfloor \texttt{Component<_> } x\ \{ps\} \rfloor \qquad \texttt{Deactivate } r_x.x \in T \qquad \vdash^{\text{deactiv}}_{\text{prop}} (ps_i)_{r_x.x} \Downarrow \texttt{Deactivate } y}{E,A,T,E',A' \vdash^{\text{proc}}_{\text{prop}} p \Downarrow \texttt{Deactivate } y}\ \text{PC}_2$$

Figure 8: Propagation rules for components

reaction, or when the parent becomes active again, or not at all ? We chose to eliminate this question by forbidding this case entirely, as we believe it is not useful in real-world programs. In practice, our prototype compiler statically checks that this situation can never arise.

The last set of rules, in fig. 8, describes how components propagate events to their children. Rule PC_1 specifies the propagation of activation events, while rule PC_2 specifies the propagation of deactivation events. Each is specified by a judgment $\vdash^{\text{(de)activ}}_{\text{prop}} p \Downarrow ev$, which follows the same logic as the judgment for initial activation. The only difference between these two sets of rules is that activation is only propagated to children for components/bindings marked with `<a>`, while deactivation is always propagated.

3.3.2 Safe reactions

The propagation rule by itself is not enough to ensure the right semantics is given to propagation of events. Indeed, consider the pair of processes `a: 0 =: y; (x/y > 10) -> t` and suppose that assignment `a` is triggered. According to our semantics, it generates an event `Assign 0 y`. In turn, this triggers the conditional binding, as the value of `y` as changed. However, the behavior of `x/y` is undefined because it divides by 0. This means that the predicate for expression evaluation does not apply, and therefore the premises of the rule for conditional left-hand side activation does not hold. In the end, our semantics say that this program evaluates fine, but does not trigger `t`. This is wrong: actually, if we compile and/or execute this program, it will crash. Therefore, this program should not admit a semantics at all. To

$$\boxed{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} lhs}$$

$$\frac{}{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} x}\ \text{SLT} \qquad \frac{}{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} \texttt{A?(}x\texttt{)}}\ \text{SLA} \qquad \frac{}{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} \texttt{D?(}x\texttt{)}}\ \text{SLD} \qquad \frac{}{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} \texttt{C?(}x\texttt{)}}\ \text{SLC}$$

$$\frac{\forall x \in \text{free}(e),\ \forall v,\ \texttt{Assign v}\ x \notin T}{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} \texttt{(}e\texttt{)?}}\ \text{SLI}_1 \qquad \frac{E,E' \vdash_{\text{exp}} e \Downarrow v \qquad v \in \{\texttt{true},\texttt{false}\}}{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} \texttt{(}e\texttt{)?}}\ \text{SLI}_2$$

$$\boxed{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} p_r}$$

$$\frac{}{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} (ty\ x\ e)_{r_x}}\ \text{SP} \qquad \frac{}{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} (\texttt{Spike}\ x)_{r_x}}\ \text{SS} \qquad \frac{\texttt{Trigger}\ r_x.x \in T \implies E,E' \vdash_{\text{exp}} e \Downarrow v}{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} (x\texttt{:}\ e\ \texttt{=:}\ y)_{r_x}}\ \text{SA}$$

$$\frac{r_x.x \notin A'}{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} (x\texttt{:}\ lhs\ \texttt{->\<\_>}\ rhs)_{r_x}}\ \text{SB}_1 \qquad \frac{E,T,E' \vdash^{\text{lhs}}_{\text{safe}} lhs}{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} x\texttt{:}\ lhs\ \texttt{->\<\_>}\ rhs}\ \text{SB}_2$$

$$\frac{r_x.x \notin A'}{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} (\texttt{Component<\_>}\ x\ \{ps\})_{r_x}}\ \text{SC}_1 \qquad \frac{\forall i,\ E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} ps_{i_{r_x.x}}}{E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} (\texttt{Component<\_>}\ x\ \{ps\})_{r_x}}\ \text{SC}_2$$

Figure 9: Safe reaction rules

correct for these cases, we introduce an additional premise in fig. 6, $E,T,E',A' \vdash^{\text{proc}}_{\text{safe}} p_r$, which ensures that all expressions that need to be evaluated in the process p rooted at r evaluate without undefined behavior.

The definition of this judgment is presented in fig. 9. Execution of a process that does not propagate events (property, `Spike`) is always safe. An assignment is safe if, when triggered, the evaluation of its expression is safe (rule SA). Both bindings and components are safe if they are inactive (rules SB_1 and SC_1). If an assignment is active, then its left-hand side must be safe: if the left-hand side is a condition, either there is no assignment to its free variables, in which case it does not need to be evaluated (rule SLI_1), or it evaluates to a boolean value (rule SLI_2). An active component is safe if all its children processes are safe (rule SC_2).

3.3.3 State update

The two judgments described above specify what events are emitted during the reaction. It remains to specify how the state of the reactive system is updated by these events. This is the role of the judgment $E,A \vdash_{\text{update}} T \Downarrow E',A'$, which specifies that "when it receives events T, the state of a process updates from (E, A) to (E', A')". The unique rule specifying this judgment is presented in fig. 10.

Its first two premises specify the updated environment E': for each path p, either there is an `Assign` event to p, in which case $E'(x)$ is set to its value, or there is none, in which case $E'(x)$ keeps the same value as $E(x)$. The first premise implies a strong constraint of Smalite programs, that was not explicit until now: two assignments to the same property with different values may not occur during the same reaction. This is a stronger constraint than was chosen in the original implementation of Smala (where the value of the "final" assignment was kept), but it simplifies the semantic model and facilitates the generation of simple and efficient imperative code. In practice, this semantic constraint is satisfied for

$$\frac{\begin{array}{c}\forall p\ v,\ \texttt{Assign}\ \mathrm{v}\ p \in T \implies E'(p) = \lfloor v \rfloor \qquad \forall p,\ (\forall v,\ \texttt{Assign}\ \mathrm{v}\ p \notin T) \implies E'(p) = E(p) \\ \forall p,\ \texttt{Activate}\ p \in T \implies p \in A' \qquad \forall p,\ \texttt{Deactivate}\ p \in T \implies p \notin A' \\ \forall p,\ (\texttt{Activate}\ p \notin T \wedge \texttt{Deactivate}\ p \notin T) \implies (p \in A' \iff p \in A)\end{array}}{E, A \vdash_{\text{update}} T \Downarrow E', A'}\ \mathrm{U}$$

$E, A \vdash_{\text{update}} T \Downarrow E', A'$

Figure 10: State update rule

any program that respects a Reactive Static Single Assignment (RSSA) property: each reaction only contains one assignment to a given property. This property is checked by our prototype compiler.

The remaining premises specify the updated activation A': an `Activate` p event adds p in A', a `Deactivate` p event removes it. If neither type of events are emitted, then p keeps its activation status. These premises imply a second constraint: `Activate` p and `Deactivate` p events may not occur during the same reaction. Like the previous constraint, this choice simplifies both the semantics and compilation scheme, and is checked statically during compilation.

3.3.4 Discussion

Going back to the rule that composes them in fig. 6, we can now get a high-level view of how these three judgments interact interact to specify how a Smalite process reacts to an event. As discussed, the *prop* and *safe* rules specify which events are produced, while the *update* rule specifies how state is updated. These definitions appear to be mutually recursive. On the one hand, in *prop*, the updated state E', A' is used as argument to compute generated events. On the other hand, in *update*, the set of events T is used as an argument to compute the updated state E', A'. To make these semantics executable, one would most likely need to implement them as a pair of mutually-defined fixed-points.

4 Conclusion and Future work

In this paper, we have presented Smalite, a minimal language capable of encoding more fully-featured UIDLs such as Smala. The denotational semantics of the language, along with a prototype compiler, have been implemented in the Rocq prover: this is a first stepping stone towards a verification framework for UIDLs in general and Smala in particular. We propose here some leads for future work.

4.1 A formally verified compiler for Smalite

We have implemented a prototype compiler for Smalite programs that generates C code which can be linked against the SDL library [22] to define reactive GUI programs. This compiler generates executable code for Smalite programs by finding a static scheduling of instructions that implement events (setting the value of property, (un)setting activation flag), guarded by conditions that emulate the left-hand side of bindings. This involves three major passes between separate Intermediate Representations (IRs):

1. Explicitly compute the event propagation graph of the program, where vertices are instructions and edges are guarded by conditions that correspond to the left-hand side of bindings

2. Checking that this graph does not contain any cycle, and flattening it by transforming its transitive closure into direct associations between external events and lists of guarded instructions
3. For each external event, schedule the guarded instructions according to dependencies (e.g. an assignment to `x` must be processed before an instruction that depends on `x`), and generate a function that implements the reaction to the event

Our prototype compiler targets the Obc IR of the Vélus compiler [5]. Obc is an imperative object-oriented language where each class has fields and methods. In our compiler, we use fields to store the values of properties and the activation state of permanent processes, and generate one method for each external event. Then, we reuse the Obc-to-Clight pass of Vélus to produce a Clight program. Clight is one of the frontend language of the CompCert verified compiler [12] on which Vélus, and therefore our prototype compiler, are based.

In the future, we hope to reuse the correctness proofs of Vélus and CompCert to build an end-to-end semantics preservation proof from the denotational semantics presented in this paper down to the assembly semantics provided by CompCert. The missing piece is, of course, a correctness proof that relates our denotational semantics for Smalite to the operational semantics of Obc [5, §4.1.2]. We expect this proof to be difficult, for three reasons. First, our denotational semantics model is clearly more abstract than the operational semantics used for Obc: the former asserts the existence of objects on which a set of relations hold, while the latter describes precisely how these objects are computed. Second, the correctness of our compilation scheme heavily depends on the well-formedness of the source program (absence of dependency loops, absence of contradictory assignments, etc.) which might be difficult to specify precisely and reason about. Last, the rule for reactions described in fig. 6 relies on a complex predicate to define the content of the set of events T. We are afraid that reasoning about this predicate in semantics preservation proofs will require proving implications in two directions (source-to-target and target-to-source), which might incur a significant amount of work for each compilation pass.

4.2 Static analysis, simplifications and optimizations

Our prototype compiler is, in some respects, very naïve, and future work will focus on improving it so that it generates more efficient imperative code, and accepts a larger set of source Smalite programs.

Indeed, compilation to imperative code with a fixed schedule places a restriction on which programs may be accepted: some programs with well-defined and deterministic semantics may be rejected during compilation. For instance, consider a program with only two bindings: `x -> y; y -> x`. In theory, this program is not schedulable, as there is a dependency loop between the triggering of `x` and `y`. In practice however, the semantics of this program are clear: either `x` and `y` are both triggered, or they both are not. To generate imperative code, we need to somehow cut the dependency cycle, but it is not obvious how to do so in general.

Another, more complex loop is actually showcased in our example, on `lines 29-30`. `Line 29` specifies that releasing `btn2` sets `count` back to `3`. `Line 30` specifies that changing `count` triggers a test that activates `btn2` if `count < 3`. How should these two bindings be scheduled? On the one hand, before checking the first binding, it must be determined whether or not `btn2` is active, so `line 30` should be scheduled before `line 29`. On the other hand, `line 29` sets `count`, so it should be scheduled before `line 30` that listens to a change on `count`. This looks like a dependency loop, but on closer inspection the second scheduling is never useful: indeed, `line 29` sets `count` to `3`, therefore, `count < 3` will never be true on a cycle where this assignment is activated. A general way to filter this type of false dependency loops would be to "cut" chains of bindings that include conditions that will never evaluate

to true. To do so, we could use some type of static analysis such as abstract interpretation [6]. Moreover, statically simplifying conditions and cutting useless bindings would also make the generated code more efficient.

4.3 Extending Smalite

The example presented in section 2 could be simplified with a few higher-level constructs. First, we see a very similar code pattern of declaring an assignment and having a unique binding to that assignment at several places (`lines 14, 15, 17, 29, ...`). These could be simplified with a meta-process that encodes "binding to an assignment". In Smala, this is implemented by assignment sequences, with syntax `press -> { 255 =: green }`. This feature could be added as part of a more general source language than the one we present in this paper, and compiled down to the simpler constructs of Smalite.

Another unwelcome repetition appears in the definitions of the buttons `btn1` and `btn2`, which are essentially identical bar a few parameter. The full Smala language allows the user to define parameterized components which can then be instantiated in more complex programs. It is not yet clear what would be the best way to treat such a feature in our formalization: either user-defined components could be inlined into a single Smalite program, or they could be compiled separately, making Smalite and the other intermediate representations more complex.

4.4 Graphical semantics and properties

The semantic model we propose in this paper describes how the internal state of a program is updated in reaction to an event. It does not specify how this internal state is related to the observable behavior of the program. In particular, all the components used in the example (`Frame`, `Rectangle`, etc) have graphical semantics: their activation, and the value of their property affects what is displayed on the screen. Furthermore, the events that are modeled by spikes (`close`, `released`, etc) correspond to user actions. The structure of graphical components could be formalized as a scene graph, while the behavior of interactions could be specified using low-level events (click on the mouse at specific coordinates, etc). Modeling these graphical and interactive semantics at the level of the Smala source language would facilitate two high-level goals.

First, mechanize a compilation correctness proof for the code that binds the reactive program to the system libraries that implement user interactions; for our prototype compiler, that would be SDL3. Writing this proof would first require axiomatizing the behavior of all the SDL3 functions in use: drawing functions would affect the scene graph, while event-listening functions would be related to low-level events.

Second, it would be possible to reason on graphical properties of the system. Indeed, many safety-critical interactive systems, such as airplane cockpit GUIs are bound by strict norms. For instance, the ED 143 [9] specifies the behavior of the Traffic Alert and Collision Avoidance System (TCAS), which prevents aircraft from crashing into each other by alerting the pilot of imminent collisions and requesting altitude change. In particular, it specifies how incoming aircraft should be displayed on the screen.

In [21], the authors present a technique to formally verify that this specification holds for a given implementation, by generating Weakest Preconditions (WPs) and verifying them using the Z3 SMT Solver [8]. We could ground these results by proving the correspondence of the WP generation algorithm with our semantics model, and using a tool such as SMTCoq [3] to transport the proof of correctness generated by Z3 into Rocq logic. This would give us a mechanized Rocq proof that these properties hold

for the source program and, by applying the compiler correctness proof mentioned in section 4.1, that they hold for the generated binary.

Another approach would be to reuse the work proposed in [20, 19], which proposes a denotational semantic model based on bigraphs for a UIDL very similar to ours. It then uses the PRISM model checker [11] to prove that the resulting bigraphical reactive system implies graphical properties of interest. To take advantage of these results, we would first need to prove that our own relational semantic model is equivalent to the bigraphical model.

References

[1] Govindaraj Anand et al. "Avionics display for Two-Seated aircraft using OpenGL". In: *Artificial Intelligence and Evolutionary Computations in Engineering Systems: Computational Algorithm for AI Technology, Proceedings of ICAIECES 2020*. Springer. 2022, pp. 255–265. DOI: 10.1007/978-981-16-2674-6_20.

[2] Philippe Antoine and Stéphane Conversy. "Volta: The First All-Electric Conventional Helicopter". In: *MEA 2017, More Electric Aircraft*. Bordeaux, France, Feb. 2017. URL: https://enac.hal.science/hal-01609233.

[3] Michael Armand et al. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses". In: *Certified Programs and Proofs*. Vol. 7086. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 135–150. DOI: 10.1007/978-3-642-25379-9_12.

[4] Timothy Bourke, Basile Pesin, and Marc Pouzet. "Verified Compilation of Synchronous Dataflow with State Machines". In: *ACM Trans. Embed. Comput. Syst.* 22 (5s Sept. 9, 2023), 137:1–137:26. DOI: 10.1145/3608102.

[5] Lélio Brun. "Mechanized Semantics and Verified Compilation for a Dataflow Synchronous Language with Reset". PhD thesis. PSL Research University, June 2020. URL: https://theses.hal.science/tel-03068862/.

[6] Patrick Cousot and Radhia Cousot. "Static determination of dynamic properties of programs". In: *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, 1976, pp. 106–130.

[7] Mathieu Cousy et al. "AEON: Toward a concept of operation and tools for supporting engine-off navigation for ground operations". In: *Toward Sustainable Aviation Summit (TSAS) 2022*. Toulouse, France, Oct. 2022, pp. 1–12. URL: https://enac.hal.science/hal-03859908.

[8] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[9] EUROCAE. *ED 143 - Minimum Operational Performance Standards For Traffic Alert And Collision Avoidance System II (TCAS II)*. URL: https://standards.globalspec.com/std/1609213/eurocae-ed-143.

[10] Stephan Huber, Johanna Gramlich, and Tobias Grundgeiger. "From Paper Flight Strips to Digital Strip Systems: Changes and Similarities in Air Traffic Control Work Practices". In: *CSCW*. New York, NY, USA: Association for Computing Machinery, May 2020. DOI: 10.1145/3392833.

[11] M. Kwiatkowska, G. Norman, and D. Parker. “PRISM 4.0: Verification of Probabilistic Real-time Systems”. In: *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*. Ed. by G. Gopalakrishnan and S. Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 585–591. DOI: 10.1007/978-3-642-22110-1_47.

[12] Xavier Leroy. “Formal Verification of a Realistic Compiler”. In: *Communications of the ACM* 52.7 (July 2009), pp. 107–115. DOI: 10.1145/1538788.1538814.

[13] Yixiang Lim et al. “Avionics Human-Machine Interfaces and Interactions for Manned and Unmanned Aircraft”. In: *Progress in Aerospace Sciences* 102 (2018), pp. 1–46. DOI: 10.1016/j.paerosci.2018.05.002.

[14] Magnus Madsen, Ondřej Lhoták, and Frank Tip. “A Semantics for the Essence of React”. In: *LIPIcs, Volume 166, ECOOP 2020* 166 (2020). Ed. by Robert Hirschfeld and Tobias Pape, 12:1–12:26. DOI: 10.4230/LIPICS.ECOOP.2020.12.

[15] Mathieu Magnaudet et al. “Djnn/Smala: A Conceptual Framework and a Language for Interaction-Oriented Programming”. In: *Proc. ACM Hum.-Comput. Interact.* 2 (EICS June 19, 2018), 12:1–12:27. DOI: 10.1145/3229094.

[16] Ingo Maier, Tiark Rompf, and Martin Odersky. *Deprecating the observer pattern*. Tech. rep. EPFL-REPORT-148043. Lausanne: EPFL, 2010.

[17] Alice Martin, Mathieu Magnaudet, and Stéphane Conversy. “Causette: user-controlled rearrangement of causal constructs in a code editor”. In: *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 2022, pp. 241–252. DOI: 10.1145/3524610.3527885.

[18] Brad A. Myers. “Separating application code from toolkits: Eliminating the spaghetti of callbacks”. In: *Proceedings of the 4th annual ACM symposium on User interface software and technology*. 1991, pp. 211–220. DOI: 10.1145/120782.120805.

[19] Nicolas Nalpon. “Vers la vérification des langages de description d'interface utilisateur”. PhD thesis. INSA de Toulouse, Mar. 13, 2023. URL: https://theses.hal.science/tel-04139613.

[20] Nicolas Nalpon, Cyril Allignol, and Celia Picard. “Towards a User Interface Description Language Based on Bigraphs”. In: *Theoretical Aspects of Computing – ICTAC 2022*. Ed. by Helmut Seidl, Zhiming Liu, and Corina S. Pasareanu. Vol. 13572. Cham: Springer International Publishing, 2022, pp. 360–368. DOI: 10.1007/978-3-031-17715-6_23.

[21] Daniel Prun and Pascal Béger. “Formal Verification of Graphical Properties of Interactive Systems”. In: *Proceedings of the ACM on Human-Computer Interaction* 6 (EICS June 14, 2022), pp. 1–30. DOI: 10.1145/3534521.

[22] Simple DirectMedia Layer project. *SDL Wiki*. URL: https://wiki.libsdl.org/SDL3/FrontPage.

[23] The Coq Development Team. *The Coq Proof Assistant*. Version 8.20. Zenodo, Sept. 4, 2024. DOI: 10.5281/zenodo.14542673.

Bisimilarity and Simulatability of Processes Parameterized by Join Interactions

Clemens Grabmayer
GSSI
L'Aquila, Italy
clemens.grabmayer@gssi.it

Maurizio Murgia
GSSI
L'Aquila, Italy
maurizio.murgia@gssi.it

Departing from Larsen's concept of parameterized bisimilarity of processes with respect to interaction with environments, we start an exploration of its natural weakening: bisimilarity of unrestricted join interactions with environments. Parameterized bisimilarity relates processes p and q with respect to an environment e if p and q behave bi-similarly while joining—respectively the same—transitions from e. The weakened variant relates processes p and q with respect to environment e if the join-interaction processes $p \,\&\, e$ and $q \,\&\, e$ of p and q with e are bisimilar. (Hereby join interactions $r \,\&\, f$ facilitate a step with label a to $r' \,\&\, f'$ if and only if r and f permit a-steps to r' and f', respectively.)

Join-interaction parameterized (ji-parameterized) bisimilarity coincides with parameterized bisimilarity for deterministic environments, but that it is a coarser equivalence in general. We explain how Larsen's concept can be recovered from ji-parameterized bisimilarity by 'determinizing' interactions. We show that by adaptation to simulatability (simulation preorder) the same concept arises: parameterized simulatability coincides with ji-parameterized simulatability. For the discrimination preorder of (ji-)parameterized simulatability on environments we obtain the same result as Larsen did for parameterized bisimilarity. Also, we give a modal-logic characterization of (ji-)parameterized simulatability. Finally we gather open problems, and provide an outlook on our current related work.

1 Introduction

With the motivation of developing flexible formal methods for proving correctness of software programs incrementally, by showing compositional correctness under the formation of contexts, Larsen in [11, 12] introduced parameterized bisimilarity of processes as a helpful concept. It turned out, more recently, to be useful in an area with a similar motivation: contextual behavioural metrics (see work of Dal Lago and Murgia [6, 10]), which measure differences between programs as distances by means of pseudo-metrics. This is because parameterized bisimilarity provides natural examples of contextual behavioural metrics.

The idea underlying parameterized bisimilarity is that the behaviors of two processes are compared with respect to a third process that represents a common environment, in which both processes are placed, and with which both can interact separately. The environment is able to 'consume' a transition from a process by performing a transition with the same action label, after which both the process and the environment move to the target state of the interaction transition on their side, respectively. Such consumption interactions are intended to continue as long as possible. Yet in case that an environment state permits no transition with the same label as the current process state (this is the case, for example, if the environment or the considered process is in a deadlock state), the consumption process stops. Given this setup, processes p and q are called bisimilar with respect to an environment process e if p and q behave in a bisimilar way (fulfilling forth and back conditions as typical for bisimulations) for any pair of runs of *synchronous* consumption interactions of the environment with the two processes in which the environment takes the *same* transitions on its side.

C. Aubert, C. Di Giusto, S. Fowler & V. K. I Pun (Eds.):
18th Interaction and Concurrency Experience (ICE 2025)
EPTCS 425, 2025, pp. 36–54, doi:10.4204/EPTCS.425.4

It is distinctive for Larsen's concept of parameterized bisimilarity that the forth and back conditions of two processes p and q, and subsequently, of states reached via transitions from p and q, have to be verified *separately* for every run of the environment but while interacting *synchronously* with both processes. Indeed, comparisons of possible further interactions have to be carried out in synchronicity of the interactions, as long as the environment can interact with either of the processes. Thereby a mismatch is detected in the following situation: Suppose that by successful comparisons in a synchronous run derivative processes p' and q' as well as derivative environment e' are reached. Suppose further that e' permits, say, an a-transition that can be joined only with an a-transition from p', but not from q' (in case q' does not permit a-transitions). Then it has been determined that p and q are not bisimilar with respect to e.

Parameterized bisimilarity thus compares the behavior of two processes with respect to *controlled* and *synchronous* interactions with an environment process. For determining whether two processes p and q are bisimilar with respect to an environment process e it is necessary to observe the consumption interactions of p with e and of q with e in a synchronous step-wise manner. It is not sufficient to be merely presented the completed processes that result from the interactions of p with e, and of q with e, respectively, and then to ask whether these results are bisimilar.

There are, however, conceivable practical situations, in which one lacks sufficient control over the environment process in order to perform, or merely to analyze, controlled and synchronous interactions with the considered processes. That is, situations in which a scientist has access only to the data of completed interactions of two processes with a given environment, but in which she lacks sufficient control over the environment in order to perform the two interactions synchronously in a step-by-step manner so that she can compare the behaviors that remain after each step.

Here we define, and start to investigate, the weaker concept of parameterized bisimilarity in which only the completed outcome processes of the possible interactions of two processes p and q with a given environment are compared as to whether they are bisimilar. For this purpose we stipulate that the consumption interaction takes place in the form of a 'join' operation ($\&$) between each process and the environment, which produces transitions with the same action labels as the two interaction transitions. Indeed, only transitions with the same label from a process and the environment can be 'joined' to interact, and produce a resulting transition with again that same label. We call the concept of bisimilarity between the join interactions of each process with the environment 'join-interaction parameterized' (ji-parameterized) bisimilarity. Larsen briefly mentions this concept at the end of the article [12]. He calls it 'perhaps more immediate', but excludes it from further consideration on the basis that it lacks some distinctive properties that he was able to show for parameterized bisimilarity. (For more details, see the paragraph 'Larsen on ji-parameterized bisimilarity ...' in Section 5.) Although that is true, it remains the case that ji-parameterized bisimilarity has a much easier, and appealingly natural definition, and that it may be of practical use in cases in which parameterized bisimilarity cannot be used.

While these considerations may seem abstract, we got interested in studying ji-parameterized bisimilarity when we made the following concrete observations (many of which are explained here later):

- parameterized bisimilarity and ji-parameterized bisimilarity do not coincide, so we try to understand the conceptual difference between them, also by means of concrete examples (for an overview see Theorem 3.11);
- noticing that, for deterministic environments, parameterized bisimilarity and ji-parameterized bisimilarity coincide (see Proposition 3.10);
- recognizing that also parameterized bisimilarity can be formulated as bisimilarity of a special kind ($\&_\bullet$) of join interaction (see Definition 3.4 and Lemma 3.5);
- recognizing that simulation preorder adaptations of the two concepts of parameterized bisimilarity

and ji-parameterized bisimilarity do in fact coincide (see Proposition 3.10, (i));

- Larsens modal-logical characterization of parameterized bisimilarity for (ji-)parameterized can be easily adapted to simulatability[1] (see Theorem 4.2 in Section 4);
- a natural modal-logical characterization also for ji-parameterized bisimilarity is worth studying, albeit we do not provide a solution to it in this work (see current work item (W1) in Section 5).

In Section 2 we summarize Larsen's definition and main results on parameterized bisimilarity, and we define parameterized simulatability.[1] In Section 3 we define ji-parameterized bisimilarity and simulatability, and develop basic results about their relationships with parameterized bisimilarity and simulatability. We discover that Larsen's theorem about the discrimination preorder induced by parameterized bisimilarity has an analogous version for the discrimination preorder that is induced by (ji-)parameterized simulatability. Then in Section 4 we specialize Larsen's modal-logical characterization of parameterized bisimilarity to (ji-)parameterized simulatability. Finally in Section 5 we give a list that summarizes our results, we report about the literature and our ongoing related work, and we sketch further ideas and plans.

2 Preliminaries on Larsen's parameterized bisimilarity

In this section we summarize definitions and results by Larsen in [11, 12] concerning parameterized bisimilarity, its induced discrimination preorder, and a modal-logical characterization for it. Additonally we define parameterized simulations, which relate to parameterized bisimulations in the same way as how simulations relate to bisimulations. We start with the basic concept of labeled transition system.

Definition 2.1 (LTSs)**.** A *(simple) labeled transition system (LTS)* is a triple $\mathscr{T} = \langle \mathrm{St}, A, \to \rangle$ that consists of a set St of *states*, a set A of *actions*, and a ternary *transition relation* $\to \subseteq \mathrm{St} \times A \times \mathrm{St}$ that represents A-labeled *transitions* on the state set.

For LTSs we will use notation and terminology for basic properties as follows. For their stipulation, we let $\mathscr{T} = \langle \mathrm{St}, A, \to \rangle$ be an LTS. For $\langle s, a, t \rangle \in \to$ we usually write $s \xrightarrow{a} t$, and say that "in state s there is a transition with label a (symbolizing an action called a) to state t". In this case we also say that t is an *a-derivative* of s. For $s \in \mathrm{St}$ and $a \in A$ we write $s \xrightarrow{a}$ if there is an a-transition from s in $\mathscr{T}$, and $s \not\xrightarrow{a}$ if there is no a-transition from s in $\mathscr{T}$.

We call an LTS $\mathscr{T} = \langle \mathrm{St}, A, \to \rangle$ *deterministic* (respectively *image-finite*) if $|\{s' \mid s \xrightarrow{a} s'\}| \leq 1$ (and respectively if $|\{s' \mid s \xrightarrow{a} s'\}| < \infty$) for all states $s \in \mathrm{St}$ and actions $a \in A$, that is, if every state of $\mathscr{T}$ has at most one a-derivative (resp. has only finitely many a-derivatives), for all $a \in A$. We say that a state $s \in \mathrm{St}$ *is deterministic* (resp. *is image-finite*) if every state of $\mathscr{T}$ that is reachable from s via a path of transitions has at most one a-derivative (resp. has only finitely many a-derivatives), for all $a \in A$.

Following well-known intuitions, bi-/simulations on such simple LTSs can be defined as follows.

Definition 2.2 (bisimulation/bisimilar, simulation/simulated by)**.** Let $\mathscr{T} = \langle \mathrm{St}, A, \to \rangle$ be an LTS.

(i) A *bisimulation B on $\mathscr{T}$* is a non-empty binary relation $B \subseteq \mathrm{St} \times \mathrm{St}$ with the following property: If $s \, B \, t$ for $s, t \in \mathrm{St}$, then the following two conditions hold:

(forth) $(\forall s' \in \mathrm{St})\big[s \xrightarrow{a} s' \implies (\exists t' \in \mathrm{St})[t \xrightarrow{a} t' \wedge s' \, B \, t']\big]$,

(back) $(\forall t' \in \mathrm{St})\big[t \xrightarrow{a} t' \implies (\exists s' \in \mathrm{St})[s \xrightarrow{a} s' \wedge s' \, B \, t']\big]$.

[1] We use 'simulatability' instead of 'similarity' for 'simulation preorder' for two reasons: to prevent the impression that a symmetrical relation were meant, and to avoid a possible confusion with 'simulation equivalence' that will also appear here.

For processes $s,t \in \mathrm{St}$, we write $s \sim t$ and say that *s and t are bisimilar* if there is a bisimulation B on $\mathscr{T}$ such that $s\,B\,t$.

(ii) A *simulation B on $\mathscr{T}$* is a non-empty binary relation $S \subseteq \mathrm{St} \times \mathrm{St}$ with the following property: If $s\,B\,t$ for $s,t \in \mathrm{St}$, then the condition (forth) in (i) holds for $B := S$ (but not necessarily the condition (back)). For processes $s,t \in \mathrm{St}$, we write $s \leq t$ and say that *s can be simulated by t*, and we write $t \geq s$ and say that *t can simulate s*, if there is a simulation S on $\mathscr{T}$ such that $s\,S\,t$.

Rather than defining simulations as weakened versions of bisimulations as above, bisimulations can also be defined from simulations, as follows. A relation B on an LTS $\mathscr{T}$ is a bisimulation if and only if both B as its converse relation $B^{\smile} := \{\langle t,s\rangle \mid s\,B\,t\}$ are simulations on $\mathscr{T}$.

For modeling processes whose behavior is studied according to how they interact with environments, both processes and environments are formalized as LTSs. However, in order to indicate their intended roles for occurring LTSs, we distinguish in notation, name, and in how they are referenced between *process LTSs* $\mathscr{P} = \langle \mathrm{Pr}, A, \rightarrow\rangle$, whose states we call *processes*, and *environment LTSs* $\mathscr{E} = \langle \mathrm{Env}, A, \Rightarrow\rangle$, whose states we call *environments*. We follow Larsen [11, 12] in this terminology and in most of the notation. Based on this distinction, Larsen defines parameterized bisimulation and bisimilarity as follows.

Definition 2.3 (parameterized bisimulation (Larsen [11, 12]))**.** Let $\mathscr{P} = \langle \mathrm{Pr}, A, \rightarrow\rangle$ be a process LTS, and let $\mathscr{E} = \langle \mathrm{Env}, A, \Rightarrow\rangle$ be an environment LTS. An *$\mathscr{E}$-parameterized bisimulation $\mathscr{B}$ on $\mathscr{P}$* is an Env-indexed family $\mathscr{B} = \{B_f\}_{f\in\mathrm{Env}}$ of non-empty binary relations $B_f \subseteq \mathrm{Pr} \times \mathrm{Pr}$ such that the following holds:[2] If $p\,B_e\,q$ for $e \in \mathrm{Env}$, then if $e \stackrel{a}{\Rightarrow} e'$ for $a \in A$ the following conditions hold:

(forth) $(\forall p' \in \mathrm{Pr})\big[\, p \stackrel{a}{\rightarrow} p' \implies (\exists q' \in \mathrm{Pr})[\, q \stackrel{a}{\rightarrow} q' \wedge p'\,B_{e'}\,q' \,]\big]$,

(back) $(\forall q' \in \mathrm{Pr})\big[\, q \stackrel{a}{\rightarrow} q' \implies (\exists p' \in \mathrm{Pr})[\, p \stackrel{a}{\rightarrow} p' \wedge p'\,B_{e'}\,q' \,]\big]$.

For processes $p,q \in \mathrm{Pr}$, and environments $e \in \mathrm{Env}$ we write $p \sim_e q$ and say that *p and q are bisimilar with respect to e* if there is an $\mathscr{E}$-parameterized bisimulation $\mathscr{B} = \{B_f\}_{f\in\mathrm{Env}}$ such that $p\,B_e\,q$.

While simulation plays a crucial role in Larsen's main theorem on parameterized bisimulation, see Theorem 2.5 below, it is surprising that he did not also define the simulation version of this concept with only the forth condition from its progression conditions. For the reason that it can be linked directly to the simulation version of the concept of 'ji-parameterized bisimulation' that we will introduce in Section 3 (see Definition 3.3), we also define 'parameterized simulation' here.

Definition 2.4 (parameterized simulation)**.** Let $\mathscr{P} = \langle \mathrm{Pr}, A, \rightarrow\rangle$ be a process LTS, and let $\mathscr{E} = \langle \mathrm{Env}, A, \Rightarrow\rangle$ be an environment LTS. An *$\mathscr{E}$-parameterized simulation $\mathscr{S}$ on $\mathscr{P}$* is an Env-indexed family $\mathscr{S} = \{S_f\}_{f\in\mathrm{Env}}$ of non-empty binary relations $S_f \subseteq \mathrm{Pr} \times \mathrm{Pr}$ such that the following holds: If $p\,S_e\,q$ holds for $e \in \mathrm{Env}$, then for all $a \in A$, if $e \stackrel{a}{\Rightarrow} e'$ the condition (forth) in Def. 2.3 for $B_{e'} := S_{e'}$ holds:

(forth) $(\forall p' \in \mathrm{Pr})\big[\, p \stackrel{a}{\rightarrow} p' \implies (\exists q' \in \mathrm{Pr})[\, q \stackrel{a}{\rightarrow} q' \wedge p'\,S_{e'}\,q' \,]\big]$.

For processes $p,q \in \mathrm{Pr}$, and environments $e \in \mathrm{Env}$ we write $p \leq_e q$ and say that *p can be simulated by q with respect to e*, and $q \geq_e p$ and say that *q can simulate p with respect to e*, if there is an $\mathscr{E}$-parameterized simulation $\mathscr{S} = \{S_f\}_{f\in\mathrm{Env}}$ such that $p\,S_e\,q$.

Also parameterized bisimulations can be defined from parameterized simulations: for process LTS $\mathscr{P}$, and environment LTS $\mathscr{E}$, $\mathscr{S} = \{S_f\}_{f\in\mathrm{Env}}$ is an $\mathscr{E}$-parameterized bisimulation on $\mathscr{P}$ if and only if $\{S_f\}_{f\in\mathrm{Env}}$, and the family $\{S_f^{\smile}\}_{f\in\mathrm{Env}}$ of converse relations of S_f are $\mathscr{E}$-parameterized simulations.

[2] Note the occurrence of e' (instead of e) in $B_{e'}$ in both of the conditions (back) and (forth).

Larsen's main result on parameterized bisimilarity concerns the *discrimination preorder* $\sqsubseteq$ that orders environments according to their power of discriminating between processes. It is defined, for a given process LTS $\mathscr{P} = \langle \mathrm{Pr}, A, \to \rangle$ and a given environment LTS $\mathscr{E} = \langle \mathrm{Env}, A, \Rightarrow \rangle$ (so that, for all $e \in \mathrm{Env}$, the relations $\sim_e$ are then fixed as subsets of $\mathrm{Pr} \times \mathrm{Pr}$), for all $e, f \in \mathrm{Env}$ by:

$$e \sqsubseteq f \;:\Longleftrightarrow\; \nsim_e \subseteq \nsim_f \qquad (\Longleftrightarrow\; \sim_f \subseteq \sim_e)\,. \tag{2.1}$$

Now Larsen's result characterizes $\sqsubseteq$ as coinciding with the simulation preorder $\leq$ on environments. For the 'completeness' direction of this characterization to hold ("$\Leftarrow$" in (2.2)), it is, however, necessary to assume that the underlying process LTS is, as Larsen formulates it, 'sufficiently rich' structurally. The weak natural assumption that he uses for the purpose of guaranteeing sufficient structural richness of any considered process LTS is that its set of processes is closed under action prefixing and finite summation (see Definition 3.2 in Section 3).

Theorem 2.5 (Larsen [11, 12])**.** *The following logical equivalence holds, provided that the underlying process LTS is closed under action prefixing and finite summation, for all image-finite environments e, f:*

$$e \leq f \iff e \sqsubseteq f \quad (\Longleftrightarrow\; \sim_f \subseteq \sim_e)\,. \tag{2.2}$$

The implication "$\Rightarrow$" holds for all (thus also for not necessarily image-finite) environments e and f.

Specifically for the direction "$\Leftarrow$" in (2.2) Larsen provides an impressive, technical proof, which he found, as he writes, only after an intensive search that took several months.

We now turn to modal-logical characterizations of the relations of being able to be simulated by $\leq$, of bisimilarity $\sim$, and of parameterized bisimilarity $\sim_e$. For expressing properties of LTSs such as the existence of a transition with label a from a given state such that at the target state property ϕ_0 holds, modal formulas should include a diamond modality $\langle a \rangle$ to build formulas like $\langle a \rangle \phi_0$. The set $\mathscr{M}$ of simple modal formulas (and the set $\mathscr{L}$ of positive formulas) are now defined with these diamond modalities and basic propositional connectives (resp. such connectives except negation) as constructors.

Definition 2.6 (modal formulas)**.** For given sets A of actions, we define the following classes of formulas: $\mathscr{L}(A)$ of positive formulas, and $\mathscr{M}(A)$ of (simple modal logic) formulas, via the following grammars:

$$\mathscr{L}(A) \qquad \phi ::= \top \mid \phi \wedge \phi \mid \langle a \rangle \phi \qquad \text{(where } a \in A\text{)}, \tag{2.3}$$

$$\mathscr{M}(A) \qquad \phi ::= \top \mid \neg\phi \mid \phi \wedge \phi \mid \langle a \rangle \phi \qquad \text{(where } a \in A\text{)}. \tag{2.4}$$

As above, we usually will keep the underlying set A of actions implicit, and write $\mathscr{L}$ and $\mathscr{M}$ for $\mathscr{L}(A)$ and $\mathscr{M}(A)$, respectively.

Definition 2.7 (satisfaction relation, sets of satisfied formulas)**.** Let $\mathscr{P} = \langle \mathrm{Pr}, A, \to \rangle$ be a process LTS. The *satisfaction relation* $\vDash \subseteq \mathrm{Pr} \times \mathscr{M}$ *on* $\mathscr{P}$ is defined by the following clauses:

$$p \vDash \top \;:\Longleftrightarrow\; p \in \mathrm{Pr}\,, \qquad p \vDash \phi_1 \wedge \phi_2 \;:\Longleftrightarrow\; p \vDash \phi_1 \text{ and } p \vDash \phi_2\,,$$

$$p \vDash \neg\phi_0 \;:\Longleftrightarrow\; p \nvDash \phi_0\,, \qquad p \vDash \langle a \rangle \phi_0 \;:\Longleftrightarrow\; \exists\, p' \in \mathrm{Pr}\,(\, p \xrightarrow{a} p' \text{ and } p' \vDash \phi_0\,)\,.$$

by induction on the structure of formulas in $\mathscr{M}$. For all processes $p \in \mathrm{Pr}$ we define by:

$$\mathscr{L}(p) := \{\phi \in \mathscr{L} \mid p \vDash \phi\}\,, \qquad \mathscr{M}(p) := \{\phi \in \mathscr{M} \mid p \vDash \phi\}\,,$$

the set $\mathscr{L}(p)$ of positive formulas in $\mathscr{L}$ that are satisfied in p, and respectively, the set $\mathscr{M}(p)$ of formulas in $\mathscr{M}$ that are satisfied in p.

The classical characterization result via modal-logical formulas of the relations bisimilarity $\sim$, and 'being able to be simulated by' $\leq$ is the following well-known theorem by Hennessy and Milner.

Theorem 2.8 (Hennessy, Milner [8]). *For all image-finite processes p and q the following statements hold:*

$$p \leq q \iff \mathscr{L}(p) \subseteq \mathscr{L}(q)\,, \tag{2.5}$$

$$p \sim q \iff \mathscr{M}(p) = \mathscr{M}(q)\,. \tag{2.6}$$

The implications "$\Rightarrow$" hold for all (thus also for not necessarily image-finite) processes p and q.

For a modal-logical characterization of parameterized bisimilarity, the concept of negation closure of positive formulas will be needed. By departing slightly from Larsen's exposition in [11, 12] we define it via a projection of general formulas to positive formulas.

Definition 2.9 (positive-formula projection, negation closure). The *positive-formula projection* is the function $|\cdot|_+ : \mathscr{M} \longrightarrow \mathscr{L}$ that maps formulas $\phi \in \mathscr{M}$ to positive formulas $|\phi|_+ \in \mathscr{L}$, and that is defined by induction on the structure of ϕ via the following clauses, for all formulas $\phi_0, \phi_1, \phi_2 \in \mathscr{M}$:

$$|\top|_+ := \top\,, \qquad |\neg\phi_0|_+ := |\phi_0|_+\,, \qquad |\phi_1 \wedge \phi_2|_+ := |\phi_1|_+ \wedge |\phi_2|_+\,, \qquad |\langle a\rangle\phi_0|_+ := \langle a\rangle|\phi_0|_+\,.$$

For every positive formula $\phi \in \mathscr{L}$, we define by $\overline{\phi}^{\neg} := \{\psi \in \mathscr{M} \mid |\psi|_+ = \phi\}$ the *negation closure of ϕ in $\mathscr{M}$*. For subclasses $\mathscr{F} \subseteq \mathscr{L}$ of positive formulas, we define the *negation closure of $\mathscr{F}$ in $\mathscr{M}$* by $\overline{\mathscr{F}}^{\neg} := \bigcup\{\overline{\phi}^{\neg} \mid \phi \in \mathscr{F}\} = \{\psi \in \mathscr{M} \mid |\psi|_+ \in \mathscr{F}\}$.

Larsen presents [11, 12] the following modal characterization theorem of parameterized bisimilarity, which he attributes to Colin Stirling. The characterization restricts consideration for possible discriminating formulas to those in the negation-closure of positive formulas that are satisfied by the environment.

Theorem 2.10 (Stirling and Larsen, [11, 12]). *For all image-finite processes p, q, and environments e:*[3]

$$\begin{aligned} p \sim_e q &\overset{(\star)}{\iff} \mathscr{M}(p) \cap \overline{\mathscr{L}(e)}^{\neg} = \mathscr{M}(q) \cap \overline{\mathscr{L}(e)}^{\neg} \\ &\iff \forall \phi_0 \in \mathscr{L}\left[e \vDash \phi_0 \;\Rightarrow\; \forall \phi \in \overline{\phi_0}^{\neg}\left(p \vDash \phi \;\Leftrightarrow\; q \vDash \phi\right)\right]. \end{aligned} \tag{2.7}$$

The implication "$\Rightarrow$" in $(\star)$ holds for all (thus also for not necessarily image-finite) p, q, and e.

3 Join-Interaction parameterized simulatability and bisimilarity

In this section we first define the weaker versions of parameterized bisimilarity and simulatability (simulation preorder) that are based on a definition of 'join-interaction' of LTSs (Definition 3.4): ji-parameterized simulatability and bisimilarity (Definition 3.3). Then we investigate the basic relationship between the new concepts and parameterized bisimilarity and simulatability (Theorem 3.11), and explain that also parameterized bisimilarity and simulatability can be viewed as bisimilarity and simulatability, resp., with respect to a special kind of join operation (Lemma 3.5). Finally we present a theorem (Theorem 3.14) that characterizes the discrimination preorder of (ji-)parameterized simulatability in analogy with Larsen's characterization of the discrimination preorder of parameterized bisimilarity, see Theorem 2.5.

[3] The condition of being image-finite can be dropped for the environments e. This can be verified by means of a careful analysis of the proof in [11, 12] for this logical characterization.

Definitions of ji-parameterized simulatablity and bisimilarity

In order to prepare for the definition of ji-parameterized bisimilarity we define 'join-interaction LTSs' by using an operation of processes that Larsen calls 'join' [11, p. 43,44]. Later we also need the subsequent stipulation of when a single LTS is closed under the 'join' operation.

Definition 3.1 (join-interaction of LTSs)**.** Let $\mathscr{T}_1 = \langle \mathrm{Pr}_1, A, \to_1 \rangle$ and $\mathscr{T}_2 = \langle \mathrm{Pr}_2, A, \to_2 \rangle$ two LTSs. By the *join-interaction of $\mathscr{T}_1$ and $\mathscr{T}_2$* we mean the LTS $\mathscr{T}_1 \,\&\, \mathscr{T}_2 = \langle \mathrm{Pr}_1 \,\&\, \mathrm{Pr}_2, A, \to \rangle$ where $\to \subseteq (\mathrm{Pr}_1 \,\&\, \mathrm{Pr}_2) \times A \times (\mathrm{Pr}_1 \,\&\, \mathrm{Pr}_2)$ with $\mathrm{Pr}_1 \,\&\, \mathrm{Pr}_2 := \{p_1 \,\&\, p_2 \mid p_1 \in \mathrm{Pr}_1,\, p_2 \in \mathrm{Pr}_2\}$ is defined via the transition system rule:

$$\frac{p_1 \xrightarrow{a}_1 p_1' \qquad p_2 \xrightarrow{a}_2 p_2'}{p_1 \,\&\, p_2 \xrightarrow{a} p_1' \,\&\, p_2'}$$

The symbol "&" in processes $p_1 \,\&\, p_2$ of $\mathscr{T}_1 \,\&\, \mathscr{T}_2$ is to be understood as a term constructor that from any two processes q_1 in Pr_1 and q_2 in Pr_2 constructs a formal join-interaction process $p_1 \,\&\, p_2$ in $\mathrm{Pr}_1 \,\&\, \mathrm{Pr}_2$.

Definition 3.2 (closure of an LTS under action prefixing, sum, and join)**.** Let $\mathscr{T} = \langle \mathrm{St}, A, \to \rangle$ be a labeled transition system. We say that $\mathscr{T}$ *is closed under action prefixing*, resp. *under sum*, and resp. *under join* if for every states $s, s_1, s_2 \in \mathrm{St}$ there exists a state $a.s \in \mathrm{St}$ for all $a \in A$, resp. there exists a state $s_1 + s_2 \in \mathrm{St}$, and resp. there exists a state $s_1 \,\&\, s_2 \in \mathrm{St}$ such that the respective transition rule below is satisfied:

$$\frac{}{a.s \xrightarrow{a} s} \qquad \frac{s_i \xrightarrow{a} s_i'}{s_1 + s_2 \xrightarrow{a} s_i'} \text{ (where } i \in \{1,2\}) \qquad \frac{s_1 \xrightarrow{a} s_1' \qquad s_2 \xrightarrow{a} s_2'}{s_1 \,\&\, s_2 \xrightarrow{a} s_1' \,\&\, s_2'}$$

We now proceed to defining join-interaction versions of parameterized simulatability, simulation equivalence, and bisimilarity as simulatability, simulation equivalence, and bisimilarity, respectively, of join-interactions between two processes and an environment.

Definition 3.3. Let $\mathscr{P} = \langle \mathrm{Pr}, A, \to \rangle$ be a process LTS, and let $\mathscr{E} = \langle \mathrm{Env}, A, \Rightarrow \rangle$ be an environment LTS.

For all environments $e \in \mathrm{Env}$, we define three binary relations on Pr: *ji-parameterized simulatability* $\leq_{\&e}$, *ji-parameterized bisimilarity* $\sim_{\&e}$, and finally, *ji-parameterized simulation equivalence* $(\leq\geq)_{\&e}$ where $\leq_{\&e}, \sim_{\&e}, (\leq\geq)_{\&e} \subseteq \mathrm{Pr} \times \mathrm{Pr}$, are defined by the following clauses, for all processes $p, q \in \mathrm{Pr}$:

(p *can be simulated by* q *with respect to join-interaction with* e)
$$p \leq_{\&e} q \;:\Longleftrightarrow\; p \,\&\, e \leq q \,\&\, e\,, \tag{3.1}$$

(p *is bisimilar to* q *with respect to join-interaction with* e)
$$p \sim_{\&e} q \;:\Longleftrightarrow\; p \,\&\, e \sim q \,\&\, e\,, \tag{3.2}$$

(p *and* q *are simulation equivalent with respect to join-interaction with* e)
$$p\ (\leq\geq)_{\&e}\ q \;:\Longleftrightarrow\; p \leq_{\&e} q \ \wedge\ q \leq_{\&e} p\,, \tag{3.3}$$

where $p \,\&\, e$ and $q \,\&\, e$ on the right in (3.1) and in (3.2) are processes from the join-interaction LTS $\mathscr{P} \,\&\, \mathscr{E}$. By $\geq_{\&e}$ we denote the converse of $\leq_e$, and express $q \geq_{\&e} p$ verbally by saying that q *can simulate* p *with respect to join-interaction with* e.

Relationship of ji-parameterized bisimilarity with parameterized bisimilarity

In order to recognize Larsen's parameterized bisimilarity as bisimilarity with respect to a specific form of join-interaction, we introduce a 'right-determinizing' variant $\&_\bullet$ of the join operation $\&$. For interactions of a process p with an environment e this operation yields the process $p \,\&_\bullet\, e$ from which transitions are labeled by pairs $\langle a, e' \rangle$ that result from joining an a-transition from p with an a-transition from e to target e'. In this way different environment steps that originally have the same action label are distinguished from

$\&_\bullet$-joins. Indeed, by making different targets of environment transitions visible as different transitions from $p\,\&_\bullet\, e$ for processes p and q and an environment e, a correspondence arises between bisimulations that link $p\,\&_\bullet\, e$ and $q\,\&_\bullet\, e$ and parameterized bisimulations that link p and q with respect to e.

Definition 3.4 (right-determinizing join-interaction with environment LTSs)**.** Let $\mathscr{P} = \langle \mathrm{Pr}, A, \to \rangle$ be a process LTS, and $\mathscr{E} = \langle \mathrm{Env}, A, \Rightarrow \rangle$ be an environment LTS. By the *right-determinizing join-interaction of $\mathscr{P}$ and $\mathscr{E}$* we understand the LTS of the form $\mathscr{P}\,\&_\bullet\,\mathscr{E} = \langle \mathrm{Pr}\,\&_\bullet\,\mathrm{Env}, A \times \mathrm{Env}, \to \rangle$ with $\mathrm{Pr}\,\&_\bullet\,\mathrm{Env} := \{p\,\&_\bullet\, e \mid p \in \mathrm{Pr}, e \in \mathrm{Env}\}$ and where $\to \subseteq (\mathrm{Pr}\,\&_\bullet\,\mathrm{Env}) \times (A \times \mathrm{Env}) \times (\mathrm{Pr}\,\&_\bullet\,\mathrm{Env})$ is defined by as transitions that are generated by the following rules:

$$\frac{p \xrightarrow{a} p' \qquad e \overset{a}{\Rightarrow} e'}{p\,\&_\bullet\, e \xrightarrow{\langle a, e' \rangle} p'\,\&_\bullet\, e'}$$

Hereby "$\&_\bullet$" in processes $p\,\&_\bullet\, e$ of $\mathscr{P}\,\&_\bullet\,\mathrm{Env}$ has to be understood as a term constructor that from any process $p \in \mathrm{Pr}$ and environment $e \in \mathrm{Env}$ constructs a formal join-interaction process $p\,\&_\bullet\, e$ in $\mathrm{Pr}\,\&_\bullet\,\mathrm{Env}$.

Now this variant "$\&_\bullet$" of the join operation "&" facilitates characterizations of parameterized simulatability and bisimilarity that are analogous in kind to the definitions of ji-parameterized simulatability and bisimilarity in Definition 3.3. As stated by logical equivalences in the following lemma, parameterized simulatability, and parameterized bisimilarity correspond to simulatability, and respectively to bisimilarity, of $\&_\bullet$-interactions between two processes and an environment. From this we obtain inclusions of parameterized simulatability and bisimilarity in ji-parameterized simulatability and bisimilarity.

Lemma 3.5. *For all processes p and q, and environments e the following two chains of statements hold:*

$$\begin{aligned} p \leq_e q &\iff (p\,\&_\bullet\, e) \leq (q\,\&_\bullet\, e) && p \sim_e q \iff (p\,\&_\bullet\, e) \sim (q\,\&_\bullet\, e) && (3.4)\\ &\implies (p\,\&\, e) \leq (q\,\&\, e) && \implies (p\,\&\, e) \sim (q\,\&\, e) && (3.5)\\ &\iff p \leq_{\&e} q\,, && \iff p \sim_{\&e} q\,, \end{aligned}$$

where $p\,\&_\bullet\, e$ and $q\,\&_\bullet\, e$ are processes from the right-determinizing join-interaction LTS $\mathscr{P}\,\&_\bullet\,\mathscr{E}$, and $p\,\&\, e$ and $q\,\&\, e$ are processes from the join-interaction LTS $\mathscr{P}\,\&\,\mathscr{E}$.

Proof (Sketch). We consider a process LTS $\mathscr{P} = \langle \mathrm{Pr}, A, \to \rangle$, and an environment LTS $\mathscr{E} = \langle \mathrm{Env}, A, \Rightarrow \rangle$.

We only argue for the chain of equivalences and implications on the right for $\sim_e$, $\sim$, and $\sim_{\&e}$, since the chain of statements on the left for $\leq_e$, $\leq$, and $\leq_{\&e}$ can be demonstrated analogously.

We first consider statement (3.4). For showing "$\Rightarrow$" it suffices to demonstrate that if $\mathscr{B} = \{B_e\}_{e \in \mathrm{Env}}$ is an $\mathscr{E}$-parameterized bisimulation on $\mathscr{P}$, then $B := \{\langle p\,\&_\bullet\, e, q\,\&_\bullet\, e \rangle \mid p\ B_e\ q\}$ is a bisimulation on $\mathscr{P}\,\&_\bullet\,\mathscr{E}$. For "$\Leftarrow$" it suffices to show that if B is a bisimulation on $\mathscr{P}\,\&_\bullet\,\mathscr{E}$, then $\mathscr{B} = \{B_e\}_{e \in \mathrm{Env}}$ with the defining clause $B_e := \{\langle p, q \rangle \mid \langle p\,\&_\bullet\, e, q\,\&_\bullet\, e \rangle \in B\}$ for $e \in \mathrm{Env}$ is an $\mathscr{E}$-parameterized bisimulation on $\mathscr{P}$. Both auxiliary statements can be shown by using the conditions (forth) and (back) from the assumed (parameterized) bisimulation in order to demonstrate the conditions (forth) and (back) of the (parameterized) bisimulation in the conclusion of the implication.

The implication in (3.5) can be easily verified similarly: by showing that whenever $B_\bullet$ is a bisimulation on $\mathscr{P}\,\&_\bullet\,\mathscr{E}$, then $B := \{\langle p\,\&\, e, q\,\&\, e \rangle \mid \langle p\,\&_\bullet\, e, q\,\&_\bullet\, e \rangle \in B_\bullet\}$ is a bisimulation on $\mathscr{P}\,\&\,\mathscr{E}$. □

In Figure 1 we illustrate the characterization (3.4) of parameterized bisimilarity $\sim_e$ as bisimilarity of $\&_\bullet$-interactions by an example. By using Lemma 3.5 we can now show that with respect to deterministic environments no difference arises between parameterized bisimilarity and ji-parameterized bisimilarity.

Proposition 3.6. *$\sim_e = \sim_{\&e}$ holds for all deterministic environments e.*

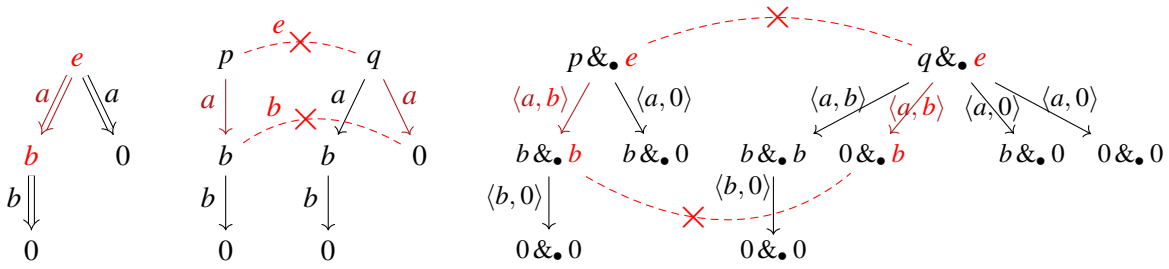


Figure 1: Example that witnesses the correspondence (3.4) in Lemma 3.5: For the environment $e := a.b + a$ and the processes $p := a.b$ and $q := e$, it holds that $p \approx_e q$ (indicated by the mismatches $\times$ when building a parameterized bisimulation on the left), and also $p \&_\bullet e \nsim q \&_\bullet e$ (indicated by the mismatches $\times$ when building a bisimulation on the right). Note that in contrast $p \sim_{\&e} q$ holds p, q, e, see Fig. 2 later.

Proof. Let $\mathscr{P} = \langle \mathrm{Pr}, A, \to \rangle$ be a process LTS, and $\mathscr{E} = \langle \mathrm{Env}, A, \Rightarrow \rangle$ be an environment LTS. The Proposition follows from Lemma 3.5, once the converse implication "$\Leftarrow$" in (3.5), which is the only implication that is missing there for $\sim_e$ to coincide with $\sim_{\&e}$, is shown to hold for deterministic environments e:

$$e \text{ is deterministic} \implies \big[(p \&_\bullet e) \sim (q \&_\bullet e) \iff (p \& e) \sim (q \& e) \big]. \tag{3.6}$$

For this, it suffices to show, that whenever B is a bisimulation on $\mathscr{P} \& \mathscr{E}$ in which all environments that occur in joins in pairs in B are deterministic, then $B_\bullet := \{ \langle p \&_\bullet e, q \&_\bullet e \rangle \mid \langle p \& e, q \& e \rangle \in B \}$ is a bisimulation on $\mathscr{P} \&_\bullet \mathscr{E}$. The assumption that only deterministic environments occur in B is not too restrictive, because derivatives of deterministic environments are deterministic again.

We let B be a bisimulation on $\mathscr{P} \& \mathscr{E}$ as described, and let $B_\bullet$ be defined as above. We have to show that $B_\bullet$ is a bisimulation on $\mathscr{P} \&_\bullet \mathscr{E}$.

For showing the condition (forth) for $B_\bullet$ to be a bisimulation on $\mathscr{P} \&_\bullet \mathscr{E}$, we let $\langle p \&_\bullet e, q \&_\bullet e \rangle \in B_\bullet$, and a transition $p \&_\bullet e \xrightarrow{l} r$ be arbitrary, where $r \in \mathrm{Pr} \&_\bullet \mathrm{Env}$, l some label in $A \times \mathrm{Pr}$. Due to operational semantics of $\mathscr{P} \&_\bullet \mathscr{E}$, this transition must actually be of the form $p \&_\bullet e \xrightarrow{\langle a, e' \rangle} p' \&_\bullet e'$, for $p' \in \mathrm{Pr}$ and $e' \in \mathrm{Env}$. We have to show that there is $s \in \mathrm{Pr} \&_\bullet \mathrm{Env}$ such that $q \&_\bullet e \xrightarrow{\langle a, e' \rangle} s$ and $\langle r, s \rangle = \langle p' \&_\bullet e', s \rangle \in B_\bullet$.

From $p \&_\bullet e \xrightarrow{\langle a, e' \rangle} p' \&_\bullet e'$ it follows that $p \xrightarrow{a} p'$ and $e \overset{a}{\Rightarrow} e'$. By the definition of $\mathscr{P} \& \mathscr{E}$ it follows that there is also the transition $p \& e \xrightarrow{a} p' \& e'$ in $\mathscr{P} \& \mathscr{E}$. From $\langle p \&_\bullet e, q \&_\bullet e \rangle \in B_\bullet$ it follows by the definition of $B_\bullet$ that $\langle p \& e, q \& e \rangle \in B$, and by the assumption on B also that e is deterministic. Then it follows from the condition (forth) of B as a bisimulation on $\mathscr{P} \& \mathscr{E}$ that there is some $s \in \mathrm{Pr} \& \mathrm{Env}$ such that $q \& e \xrightarrow{a} s$ and $\langle p' \& e', s \rangle \in B$. By the definition of $\mathscr{P} \& \mathscr{E}$ we find that $s = q' \& e''$ and $\langle p' \& e', q' \& e'' \rangle \in B$ for some $q' \in \mathrm{Pr}$ and $e' \in \mathrm{Env}$ with $q \xrightarrow{a} q'$ and $e \overset{a}{\Rightarrow} e''$. But now from $e \overset{a}{\Rightarrow} e'$ and $e \overset{a}{\Rightarrow} e''$ we can conclude, because e is deterministic, that $e'' = e'$. From this we obtain $\langle p' \& e', q' \& e' \rangle \in B$, which entails $\langle p' \&_\bullet e', q' \&_\bullet e' \rangle \in B_\bullet$. From $e \overset{a}{\Rightarrow} e'$ and $e \overset{a}{\Rightarrow} e'$ we also obtain $q \&_\bullet e \xrightarrow{\langle a, e' \rangle} q' \&_\bullet e'$. Therefore we have found in $s := q' \&_\bullet e'$ the desired $s \in \mathrm{Pr} \&_\bullet \mathrm{Env}$ with $q \&_\bullet e \xrightarrow{\langle a, e' \rangle} s$ and $\langle r, s \rangle = \langle p' \&_\bullet e', s \rangle \in B_\bullet$.

In this way we have established the condition (forth) for $B_\bullet$ to be a bisimulation on $\mathscr{P} \&_\bullet \mathscr{E}$. Since the condition (back) can be verified analogously, we conclude that $B_\bullet$ is indeed a bisimulation on $\mathscr{P} \&_\bullet \mathscr{E}$.

By having shown that $B_\bullet$ is a bisimulation on $\mathscr{P} \&_\bullet \mathscr{E}$ under the assumption that B is a bisimulation on $\mathscr{P} \& \mathscr{E}$ in which only deterministic environments occur, we have established (3.6), from which the proposition follows from Lemma 3.5 as argued above. □

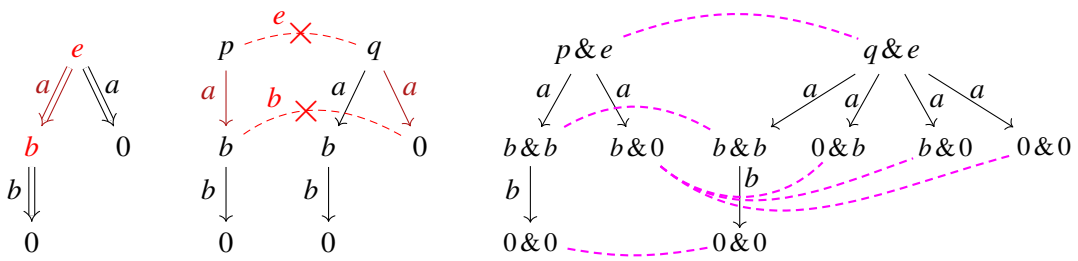


Figure 2: Example for witnessing $\sim_e \neq \sim_{\&e}$: For $e := a.b + a$, $p := a.b$, and $q := e$ it holds that $p \nsim_e q$ (indicated by the mismatches $\times$ when building a parameterized bisimulation), but $p\,\&\,e \sim q\,\&\,e$ (indicated by the bisimulation links) and hence $p \sim_{\&e} q$.

The proposition below clarifies which inclusions hold in general between $\sim_e$, $\sim_{\&e}$, and $(\leq\geq)_{\&e}$.

Proposition 3.7. *The following set-theoretical relationships hold between parameterized bisimilarity $\sim_e$, ji-parameterized bisimilarity $\sim_{\&e}$, and ji-parameterized simulation equivalence $(\leq\geq)_{\&e}$:*

(i) $\sim_e \subseteq \sim_{\&e}$ *for all environments e.*

(ii) $\sim_e \neq \sim_{\&e}$ *for some environments e, for which then* $\sim_e \subsetneq \sim_{\&e}$ *holds due to (i).*

(iii) $\sim_{\&e} \subseteq (\leq\geq)_{\&e}$ *for all environments e.*

(iv) $\sim_{\&e} \neq (\leq\geq)_{\&e}$ *for some environments e, for which then* $\sim_{\&e} \subsetneq (\leq\geq)_{\&e}$ *holds due to (iii).*

The counterexample statements (ii) and (iv) hold under the proviso that environments are included among processes, they permit at least two actions, and are closed under action prefixing and sums. This can be weakened to merely require that $a.b + a$ and $a.b$ are contained among environments and processes.

Proof. Statement (i) follows directly from the chain of implications as guaranteed by Lemma 3.5.

A counterexample for (ii) is in Figure 2: We have that $p \sim_{\&e} q$ holds due to $p\,\&\,e = (a.b+a)\,\&\,(a.b) \simeq a.b+a \sim a.b+a+a+a \simeq (a.b+a)\,\&\,(a.b+a) = q\,\&\,e$, where $\simeq$ denotes being isomorphic, which shows $p\,\&\,e \sim q\,\&\,e$. However, $p \nsim_e q$ holds for the following reason: Suppose that $p \sim_e q$ holds. Then due to $e = a.b + a \stackrel{a}{\Rightarrow} b$, and the condition (back) of an underlying parameterized bisimilarity the transition $q \stackrel{a}{\to} 0$ must be matched by the transition $p \stackrel{a}{\to} b$ so that $b \sim_b 0$ holds. However the latter is false, because $b \nsim_b 0$ holds, as the environment b and the process b can make a b-step, but 0 cannot. Statement (iii) follows from the fact that bisimilarity is symmetric and it is a simulation [20]. For (iv), let p and q be as in Figure 2, and let $e = p$. Then $p\ (\leq\geq)_{\&e}\ q$ holds due to $p\,\&\,e = (a.b)\,\&\,(a.b) \simeq a.b \leq\geq a.b + a \simeq (a.b+a)\,\&\,(a.b) = q\,\&\,e$. However, $p \nsim_{\&e} q$: indeed, $q\,\&\,e \stackrel{a}{\to} b\,\&\,0 \sim 0$, to which $p\,\&\,e$ can only answer by reducing to $b\,\&\,b \sim b$, and clearly $0 \not\sim b$. □

Parameterized simulatability coincides with ji-parameterized simulatability

While parameterized bisimilarity and ji-parameterized bisimilarity are two different relations in general by Proposition 3.7, (ii), it turns out that this does not hold for the corresponding two concepts of parameterized simulatability. For us it was surprising to find the proof of the first of the following two lemmas, which together show that parameterized simulatability and ji-parameterized simulatability coincide.

Lemma 3.8. $\leq_{\&e} \subseteq \leq_e$ *holds for all environments e.*

Proof. We fix a process LTS $\mathscr{P} = \langle \mathrm{Pr}, A, \to \rangle$, and an environment LTS $\mathscr{E} = \langle \mathrm{Env}, A, \Rightarrow \rangle$.

As the crucial stepping stone, we show that $\mathscr{S} = \{S_e\}_{e\in\mathrm{Env}}$ as defined by, for all $e \in \mathrm{Env}$:

$$S_e := \big\{ \langle p,q\rangle \in \mathrm{Pr} \;\big|\; \exists e_2 \in \mathrm{Env}\big[p\,\&\,e \;\leq\; q\,\&\,e_2\big] \big\} \subseteq \mathrm{Pr}\times\mathrm{Pr} \tag{3.7}$$

is an $\mathscr{E}$-parameterized simulation on $\mathscr{P}$. For this, we let $e \in \mathrm{Env}$, and $\langle p,q\rangle \in S_e$ be arbitrary. We assume that $e \stackrel{a}{\Rightarrow} e'$, and $p \stackrel{a}{\to} p'$ for some $a \in A$, $e' \in \mathrm{Env}$, and $p' \in \mathrm{Pr}$. We have to show that there exists $q' \in \mathrm{Pr}$ with $q \stackrel{a}{\to} q'$ such that $\langle p',q'\rangle \in S_{e'}$.

From $e \stackrel{a}{\Rightarrow} e'$ and $p \stackrel{a}{\to} p'$ we find that $p\,\&\,e \stackrel{a}{\to} p'\,\&\,e'$ holds. Due to $\langle p,q\rangle \in S_e$ we can pick $e_2 \in \mathrm{Env}$ with $p\,\&\,e \leq q\,\&\,e_2$. It follows, by the forward-property (forth) of the (largest) simulation $\leq$ applied to $p\,\&\,e \leq q\,\&\,e_2$ and $p\,\&\,e \stackrel{a}{\to} p'\,\&\,e'$, and by the operational semantics of the join operation, that there are $q' \in \mathrm{Pr}$ and $e_2' \in \mathrm{Env}$ such that $q\,\&\,e_2 \stackrel{a}{\to} q'\,\&\,e_2'$, as well as $q \stackrel{a}{\to} q'$ and $e_2 \stackrel{a}{\Rightarrow} e_2'$ and with $p'\,\&\,e' \leq q'\,\&\,e_2'$. The latter shows that $\langle p',q'\rangle \in S_{e'}$, and thus we have found $q \stackrel{a}{\to} q'$ such that $\langle p',q'\rangle \in S_{e'}$. In this way we have verified that $\mathscr{S} = \{S_e\}_{e\in\mathrm{Env}}$ as defined in (3.7) is an $\mathscr{E}$-parameterized simulation on $\mathscr{P}$.

For showing $\leq_{\&e} \subseteq \leq_e$, suppose now that $p \leq_{\&e} q$ holds, for some $p,q \in \mathrm{Pr}$ and $e \in \mathrm{Env}$. By the definition of $\leq_{\&e}$, this means that $p\,\&\,e \leq q\,\&\,e$ holds. That, however, implies $\langle p,q\rangle \in S_e$ due to (3.7). But since we have recognized $\mathscr{S}$ as an $\mathscr{E}$-parameterized simulation, we conclude that $p \leq_e q$ holds. □

Lemma 3.9. *$\leq_e \subseteq \leq_{\&e}$ holds for all environments e.*

Proof. The inclusion as stated by the lemma follows from the chain of implications displayed on the left in Lemma 3.5, which as stated in its proof can be proved analogously as the implications on the right there. But since we dropped the argument there, we also provide the sketch of a direct proof here.

Let $\mathscr{S} = \{S_e\}_{e\in\mathrm{Env}}$ be an $\mathscr{E}$-parameterized simulation on a process LTS $\mathscr{P} = \langle \mathrm{Pr}, A, \to\rangle$ with respect to an environment LTS $\mathscr{E} = \langle \mathrm{Env}, A, \Rightarrow\rangle$. Then it is easy to verify that:

$$S := \{\langle p\,\&\,e, q\,\&\,e\rangle \mid p,q \in \mathrm{Pr} \text{ and } e \in \mathrm{Env} \text{ such that } p\; S_e\; q\} \;\subseteq\; \mathrm{Pr}\,\&\,\mathrm{Env}$$

is a simulation on $\mathscr{P}\,\&\,\mathscr{E} = \langle \mathrm{Pr}\,\&\,\mathrm{Env}, A, \to\rangle$. This statement implies that if $p \leq_e q$, then $p\,\&\,e \leq q\,\&\,e$ follows, and hence $p \leq_{\&e} q$. □

Proposition 3.10. *For all environments e the following two statements hold:*

(i) $\leq_{\&e} = \leq_e$.

(ii) $(\leq\geq)_{\&e} = (\leq\geq)_e$.

Proof. The inclusions "$\subseteq$" and "$\supseteq$" that make up statement (i) are guaranteed by Lemma 3.8 and by Lemma 3.9, respectively. Then (ii) follows from (i) by: $(\leq\geq)_{\&e} = \leq_{\&e} \cap \geq_{\&e} = \leq_e \cap \geq_e = (\leq\geq)_e$. □

The theorem below collects results we have obtained about which inclusions hold in general between parameterized bisimilarity, ji-parameterized bisimilarity, and (ji-)parameterized simulation equivalence.

Theorem 3.11. *The following set-theoretical relationships hold between parameterized bisimilarity, ji-parameterized bisimilarity, and (ji-)parameterized simulation equivalence, for environments e, f, g:*

$$\begin{array}{lllllll}
\sim_e & \underset{\text{Prop. 3.7,(i)}}{\subseteq} & \sim_{\&e} & \underset{\text{Prop. 3.7,(iii)}}{\subseteq} & (\leq\geq)_e & \underset{\text{Prop. 3.10,(ii)}}{=} & (\leq\geq)_{\&e} \qquad \textit{(for \underline{all} } e\textit{)}, \\
\sim_f & \underset{\text{Prop. 3.7,(ii)}}{\subsetneq} & \sim_{\&f} & \underset{\text{Prop. 3.7,(iii)}}{\subseteq} & (\leq\geq)_f & \underset{\text{Prop. 3.10,(ii)}}{=} & (\leq\geq)_{\&f} \qquad \textit{(for \underline{some} } f\textit{)}, \\
\sim_g & \underset{\text{Prop. 3.7,(i)}}{\subseteq} & \sim_{\&g} & \underset{\text{Prop. 3.7,(iv)}}{\subsetneq} & (\leq\geq)_g & \underset{\text{Prop. 3.10,(ii)}}{=} & (\leq\geq)_{\&g} \qquad \textit{(for \underline{some} } g\textit{)},
\end{array}$$

where the statements that guarantee the relationship in question are indicated.

Discrimination preorder induced by (ji-)parameterized similarity

Larsen noted in [12, p. 209–210]: "Due to the modal characterization [see Theorem 2.10] and the simple characterization of the discrimination ordering presented [see Theorem 2.5], we are confident that the notion of parameterized bisimulation equivalence proposed is indeed a natural one". Indeed Larsen also explains that "the simulation ordering does not characterize the discrimination ordering generated by this alternative parameterized version [namely $\sim_{\&e}$]".

This is witnessed by the following proposition. Indeed it demonstrates that a characterization of the discrimination preorder induced by ji-parameterized bisimilarity $\sim_{\&e}$ cannot, in analogy with Theorem 2.5 for $\sim_e$, be of the form $e \leq f \iff \sim_{\&f} \subseteq \sim_{\&e}$, for all environments e and f.

Proposition 3.12. *There are environments e and f such that:*

$$e \leq f \ \wedge \ \sim_{\&f} \not\subseteq \sim_{\&e} . \tag{3.8}$$

Proof. Let $e = a.b$ and $f = a.b + a$. We have that $e \leq f$. Set $p = e$ and $q = f$. We have that $p \sim_{\&f} q$ but $p \not\sim_{\&e} q$, hence $\sim_{\&f}$ is not contained in $\sim_{\&e}$. □

Unfortunately we have not yet found an appealing characterization of the discrimination preorder that is induced by $\sim_{\&e}$. We formulate this question together with a perhaps also interesting specialization as the open problems (P1) and (P2) in the conclusion.

However, and somewhat surprisingly, we do obtain characterizations analogous to Theorem 2.5 for the discrimination preorder on environments e with respect to (ji-)parameterized similarity $\leq_{\&e}$ and $\leq_e$, and with respect to (ji-)parameterized simulation equivalence $(\leq\geq)_{\&e}$ and $(\leq\geq)_e$.

Similar to the proviso for the 'completeness' direction "⇒" in (2.2) of Theorem 2.5 our characterization of $(\leq\geq)_{\&e}$ requires an assumption that guarantees that the structure of the underlying process LTS is sufficiently rich in relation to the environment LTS. While Larsen assumed closure under the formation of action prefixing and finite sums, we will assume the existence of a 'universal' process, and that the underlying process LTS contains the environment LTS, and is closed under the formation of joins. (The assumption of a universal process simplifies the proof, but can be dropped.)

Let $\mathscr{P} = \langle \mathrm{Pr}, A, \to \rangle$ be a process LTS. We say that a process $u \in \mathrm{Pr}$ is *universal* if for all $a \in A$ there is a transition $u \xrightarrow{a} u'$ with $u' \in \mathrm{Pr}$ in $\mathscr{P}$ such that $u' \sim u$ (consequently it permits all actions in transitions from any of its reachable states). Note that all universal processes in $\mathscr{P}$ are bisimilar. If $\mathscr{P}$ is additionally closed under the formation &, then $u \,\&\, p \sim p \,\&\, u \sim p$ holds for all $p, u \in \mathrm{Pr}$ where u is universal.

For proving our characterization below, Theorem 3.14 we will use the following lemma.

Lemma 3.13. *For all environments e, f it holds, provided that the environment LTS is closed under joins:*

$$e \leq f \,\&\, e \iff e \leq f .$$

Proof. For showing the direction "⇒", and the direction "⇐" in the statement of the lemma, it suffices to prove that the relation $\{\langle e, f\rangle \mid e \leq f \,\&\, e\}$, and respectively, that the relation $\{\langle e, f \,\&\, e\rangle \mid e \leq f\}$ is a simulation. Both of these statements can be verified in a straightforward manner. □

We now are in a position to show characterizations of the discrimination preorders of (ji-)parameterized simulatability and (ji-)parameterized simulation equivalence, as formulated by the theorem below.

Theorem 3.14. *The following logical equivalences hold, for all environments e and f, provided that: the underlying process LTS contains a universal process and the environment LTS, and additionally is closed under the formation of joins* (but note that image-finiteness as in Theorem 2.5 is not required)*:*

$$e \leq f \iff \leq_{\&f} \subseteq \leq_{\&e} , \tag{3.9}$$

$$e \leq f \iff \geq_{\&f} \subseteq \geq_{\&e}\,, \tag{3.10}$$
$$e \leq f \iff (\leq\geq)_{\&f} \subseteq (\leq\geq)_{\&e}\,. \tag{3.11}$$

Proof. We let $\mathscr{E} = \langle \mathrm{Env}, A, \Rightarrow \rangle$ be an environment LTS, and we let $\mathscr{P} = \langle \mathrm{Pr}, A, \rightarrow \rangle$ be a process LTS that contains the universal process U.

We first note that (3.10) follows from, and is equivalent, to (3.9), because $\geq_{\&f}$ is the converse relation of $\leq_{\&e}$. Furthermore (3.11) follows from (3.9) and (3.10), due to $(\leq\geq)_{\&e} = \leq_{\&e} \cap \geq_{\&e}$. Therefore it remains to prove (3.10). For that we show the two directions of 3.9, and proceed as follows:

"$\Rightarrow$": We show that $e \leq f \implies \leq_f \subseteq \leq_e$. Then (3.9) follows from Proposition 3.10, (i). So, let $\{S_e\}_{e\in\mathrm{Env}}$ be a $\mathscr{E}$-parameterized family of binary relations $S_e \subseteq \mathrm{Pr} \times \mathrm{Pr}$ that is defined, for all $e \in \mathrm{Env}$ by:

$$p\, S_e\, q \;:\Longleftrightarrow\; \exists f \geq e \left[p \leq_f q \right]$$

It suffices to show that $\{S_e\}_{e\in\mathrm{Env}}$ is an $\mathscr{E}$-parameterized simulation. So, suppose $p\, S_e\, q$. Then $p \leq_f q$ for some $f \geq e$. Suppose $p \xrightarrow{a} p'$ and $e \stackrel{a}{\Rightarrow} e'$. Then $f \stackrel{a}{\Rightarrow} f'$ for some $f' \geq e'$, and hence $q \xrightarrow{a} q'$ for some q' such that $p' \leq_{f'} q'$. Then it follows that $p'\, S_{e'}\, q'$ holds, as required.

"$\Leftarrow$": Assume $\leq_{\&f} \subseteq \leq_{\&e}$. Let u be a universal process in $\mathscr{P}$. Then $u \,\&\, f \sim f$. Moreover, $f \leq f \,\&\, f$ (which is easy to show), and then $u \leq_{\&f} f$. By the assumption $\leq_{\&f} \subseteq \leq_{\&e}$ we have that $u \leq_{\&e} f$. In other words: $e \sim u \,\&\, e \leq f \,\&\, e$. By Lemma 3.13 we get $e \leq f$, as required. □

4 Modal characterization of (ji-)parameterized simulatability

In this section we adapt the modal characterization of parameterized bisimilarity $\sim_e$, see Theorem 2.10, for (ji-)parameterized simulatability $\leq_e$ and $\leq_{\&e}$. The crucial observation for our adaptation is Lemma 4.1 below which states that the set of positive formulas that a join interaction $p_1 \,\&\, p_2$ satisfies is the intersection of the sets of positive formulas satisfied by the constituent processes p_1 and p_2. Finally we explain why a similar line of argument is not possible in order to adapt the modal characterization for $\sim_e$ to obtain one for ji-parameterized bisimilarity $\sim_{\&e}$. In doing so we provide some evidence for Larsen's assessment, that (in view of that $\sim_e \subsetneq \sim_{\&e}$ holds in general, see Proposition 3.7) "the modal characterization for $\sim_e$ does not hold for $\sim_{\&e}$, and no other modal characterization seems immediate" [12, p. 210]. However, we report about further work of ours on this issue in the final section (see (W1) in Section 5).

Lemma 4.1. *For all processes p and q of a process LTS $\mathscr{P} = \langle Pr, A, \rightarrow \rangle$ it holds:*

$$\mathscr{L}(p \,\&\, q) = \mathscr{L}(p) \cap \mathscr{L}(q)\,. \tag{4.1}$$

Proof. Statement (4.1) can be established by induction on the structure of positive modal formulas ϕ according to their definition in grammar (2.4) of Definition 2.6.

The base case of (4.1) for $\phi = \top$ is obviously true, because $\top$ is satisfied for any process. It remains to establish the induction step for formulas of the forms $\phi = \phi_1 \wedge \phi_2$ and $\phi = \langle a \rangle \phi_0$. Since in the first case the induction step is easy to demonstrate, we only treat the more interesting case of $\phi = \langle a \rangle \phi_0$. For this we argue as follows:

$$\begin{aligned}
\langle a \rangle \phi_0 \in \mathscr{L}(p \,\&\, q) &\iff p \,\&\, q \vDash \langle a \rangle \phi_0 \\
&\iff (\exists\, p', q' \in \mathrm{Pr}) \left[p \xrightarrow{a} p' \wedge q \xrightarrow{a} q' \wedge p' \,\&\, q' \vDash \phi_0 \right] \\
&\iff (\exists\, p', q' \in \mathrm{Pr}) \left[p \xrightarrow{a} p' \wedge q \xrightarrow{a} q' \wedge \phi_0 \in \mathscr{L}(p' \,\&\, q') \right]
\end{aligned}$$

$$
\begin{aligned}
&\overset{\text{IH}}{\Longleftrightarrow} \ (\exists\, p', q' \in \mathrm{Pr}) \left[p \xrightarrow{a} p' \wedge q \xrightarrow{a} q' \wedge \phi_0 \in \mathscr{L}(p') \cap \mathscr{L}(q') \right] \\
&\Longleftrightarrow \ (\exists p' \in \mathrm{Pr}) \left[p \xrightarrow{a} p' \wedge \phi_0 \in \mathscr{L}(p') \right] \wedge (\exists q' \in \mathrm{Pr}) \left[q \xrightarrow{a} q' \wedge \phi_0 \in \mathscr{L}(q') \right] \\
&\Longleftrightarrow \ \langle a \rangle \phi_0 \in \mathscr{L}(p) \ \wedge \langle a \rangle \phi_0 \in \mathscr{L}(q) \\
&\Longleftrightarrow \ \langle a \rangle \phi_0 \in \mathscr{L}(p) \cap \mathscr{L}(q)\,,
\end{aligned}
$$

where we have marked by (IH) the logical equivalence in which the induction hypothesis is used. □

Based on this lemma, a modal characterization of $\leq_{\&e}$ and $\leq_e$ is now an easy consequence of the modal characterization of the simulation preorder $\leq$ on processes, see (2.5) in Theorem 2.8. In this way we obtain, in analogy with Theorem 2.10 , the following modal characterizations of (ji-)parameterized similarity and of (ji-)parameterized simulation equivalence with respect to positive formulas.

Theorem 4.2. *For all image-finite environments e, the following characterizations of $\leq_{\&e}$ and $\leq_e$, as well as of $(\leq\geq)_{\&e}$ and $(\leq\geq)_e$ hold for all image-finite processes p, q:*

$$
p \leq_{\&e} q \quad \big(\Longleftrightarrow \ p \leq_e q \big) \quad \Longleftrightarrow \quad \mathscr{L}(p) \cap \mathscr{L}(e) \subseteq \mathscr{L}(q) \cap \mathscr{L}(e)\,, \tag{4.2}
$$

$$
p \,(\leq\geq)_{\&e}\, q \quad \big(\Longleftrightarrow \ p \,(\leq\geq)_e\, q \big) \quad \Longleftrightarrow \quad \mathscr{L}(p) \cap \mathscr{L}(e) = \mathscr{L}(q) \cap \mathscr{L}(e)\,. \tag{4.3}
$$

The implications "$\Rightarrow$" in (4.2) *and* (4.3) *hold also for not necessarily image-finite p, q, and e.*

Proof. For (4.2) we argue as follows for all image-finite processes p and q, and environments e:

$$
\begin{aligned}
p \leq_e q \ &\Longleftrightarrow \ p \leq_{\&e} q && \text{(by Prop. 3.10, (i))} \\
&\Longleftrightarrow \ p \,\&\, e \leq q \,\&\, e && \text{(by the definition of } \leq_{\&e}) \\
&\Longleftrightarrow \ \mathscr{L}(p \,\&\, e) \subseteq \mathscr{L}(q \,\&\, e) && \text{(by (2.5) in Thm. 2.8)} \\
&\Longleftrightarrow \ \mathscr{L}(p) \cap \mathscr{L}(e) \subseteq \mathscr{L}(q) \cap \mathscr{L}(e) && \text{(by using Lem. 4.1).}
\end{aligned}
$$

The implication "$\Rightarrow$" in the third equivalence statement holds also for not necessarily image-finite p, q, and e due to the Hennessy–Milner Theorem 2.8. Together with the fact that "$\Rightarrow$" also holds for the other three equivalence statements above, this demonstrates that "$\Rightarrow$" in (4.2) holds for all p, q, and e.

Statement (4.3) for the (ji-)parameterized simulation equivalences $(\leq\geq)_{\&e}$ and $(\leq\geq)_e$ follows from (4.2) due to the definition of $(\leq\geq)_{\&e}$ from $\leq_{\&e}$ in (3.3), and of $(\leq\geq)_e$ from $\leq_e$ in Definition 2.4. □

There is no obvious generalization of Lemma 4.1 that applies to all formulas of $\mathscr{M}$. In particular, $\mathscr{M}(p_1 \,\&\, p_2) = \mathscr{M}(p_1) \cap \mathscr{M}(p_2)$ does not hold, because certainly "$\subseteq$" is violated: in case that p_1 and p_2 are such that $p_1 \xrightarrow{a}$ and $p_2 \not\xrightarrow{a}$ holds for some $a \in A$, then $\neg\langle a \rangle \top \in \mathscr{M}(p_1 \,\&\, p_2)$ due to $(p_1 \,\&\, p_2) \not\xrightarrow{a}$, but $\neg\langle a \rangle \top \notin \mathscr{M}(p_1)$, and hence $\neg\langle a \rangle \top \notin \mathscr{M}(p_1) \cap \mathscr{M}(p_2)$. Therefore the proof of the characterization above cannot be extended, at least not in an analogous manner, to obtain a modal characterization of ji-parameterized bisimilarity $\sim_{\&e}$. (But see the report about our current work (W1) in Section 5.)

Yet an interesting specialization of Lemma 4.1 concerns the specialized join operation $\&_\bullet$ introduced in Definition 3.4: $|\mathscr{M}(p \,\&_\bullet\, e)|_{\pi_A} = \mathscr{M}(p) \cap \overline{\mathscr{L}(e)}^{\neg}$ holds for all processes p and environments e, where $|\cdot|_{\pi_A}$ projects modalities $\langle\langle a, e \rangle\rangle$ in formulas to their action components $\langle a \rangle$. This observation can be used, together with the characterization of parameterized bisimilarity $\sim_e$ via $\&_\bullet$ in (3.4) of Lemma 3.5, to obtain, for Larsen's characterization of $\sim_e$ in Theorem 2.10, an alternative proof that is similar to the proof of Theorem 4.2 above.

5 Conclusion (summary, literature, current work, open problems, plans)

Here we first summarize our contributions in a list with references to statements in earlier sections. We then explain our path to the definition of ji-parameterized bisimilarity, and Larsen's comments on the shortcomings of this concept. Furthermore we collect some references to the literature concerning work that has been done based on parameterized bisimilarity in the meantime. Subsequently we report about our current work on a modal characterization of (ji-)parameterized bisimilarity, and about generalizations of the modal characterizations here and by Stirling and Larsen. We also describe some open problems of which the solutions have evaded us thus far. Finally we mention our plan to investigate whether ji-parameterized bisimilarity can be used to refine Larsen's results in his thesis [11] on a method to show program correctness under the formation of contexts.

Contribution. Below we provide a summary by listing the concepts that we have defined and the results we have obtained, together with references to the appertaining formal statements:

(C1) We complemented Larsen's parameterized bisimilarity $\sim_e$ with respect to 'synchronous' interaction with environments e by also defining parameterized simulatability, the simulation preorder $\leq_e$, on processes with respect to 'synchronous' interaction with environment e (see Definition 2.4).

(C2) We defined weaker versions $\leq_{\&e}$ of $\leq_e$ and $\sim_{\&e}$ of $\sim_e$ by relaxing the synchronicity condition of environment interaction for $\leq_e$ and $\sim_e$ to require only the existence of simulations, and respectively of bisimulations, between free join interactions (&) with environments e (see Definition 3.3).

(C3) We showed that $\leq_e$ and $\sim_e$ can be characterized similarly to the definitions of $\leq_{\&e}$, and $\sim_{\&e}$ via join interactions (&) as the existence of a simulation, and as bisimilarity, respectively, of free interactions with the specific form $\&_\bullet$ (see Definition 3.4) of join interactions that record targets of environment transitions in action labels (see Lemma 3.5).

(C4) We established that $\sim_e$ and $\sim_{\&e}$ coincide for deterministic environments e (see Proposition 3.6).

(C5) We settled the relationships between (ji-)parameterized bisimilarity $\sim_e$ and $\sim_{\&e}$, and the (ji-)parameterized simulation equivalences $(\leq\geq)_e$ and $(\leq\geq)_{\&e}$: for all environments $\sim_e$ is contained in $\sim_{\&e}$, and furthermore $\sim_{\&e}$ is contained in both of $(\leq\geq)_e$ and $(\leq\geq)_{\&e}$, which coincide. The two inclusions in this chain are proper in general. (See Theorem 3.11).

(C6) Larsen's main technical result about $\sim_e$ (see Theorem 2.5), that the discrimination preorder induced by $\sim_e$ on environments coincides with the simulation preorder $\leq$ on environments, does not hold analogously for the discrimination preorder induced by $\sim_{\&e}$ (see Proposition 3.12). However, we showed that this coincidence with the simulation preorder $\leq$ on environments *does* hold analogously for the discrimination preorders induced both by (ji-)parameterized similarity $\leq_e = \leq_{\&e}$ and by (ji-)parameterized simulation equivalence $(\leq\geq)_e = (\leq\geq)_{\&e}$ (see Theorem 3.14).

(C7) We adapted Stirling and Larsen's modal characterization of parameterized bisimilarity $\sim_e$ (see Theorem 2.10) to obtain a modal characterization of (ji-)parameterized similarity $\leq_e = \leq_{\&e}$ and also of (ji-)parameterized simulation equivalence $(\leq\geq)_e = (\leq\geq)_{\&e}$ (see Theorem 4.2).

Larsen on ji-parameterized bisimilarity, and our way to its definition. We formulated ji-parameterized bisimilarity and ji-parameterized simulatability while reading Larsen's article [12] from 1987, and trying to improve our intuitive understanding of parameterized bisimilarity. Afterwards we developed, in stages, the results that we report here. Only when diving deeper into the intricate proof of Larsens main result, the characterization of the discrimination preorder induced by parameterized bisimilarity $\sim_e$ as simulatability of environments (see Theorem 2.5), did we find his remarks about an "alternative and perhaps more immediate parameterized version [of bisimulation equivalence]". This passage appears on

page 210 in [12], at the end of Section 5 that is devoted to this central result. The version of bisimulation equivalence that Larsen sketches there coincides with ji-parameterized bisimilarity $\sim_{\&e}$.

Larsen refers to ji-parameterized bisimilarity in order to "give further support for the proposed parameterized version of bisimulation equivalence", in addition to the following assessment: "Due to the modal characterization presented [...] and the simple characterization of the discrimination ordering presented [...], we are confident that the notion of parameterized bisimulation equivalence proposed is indeed a natural one." As for the mentioned further evidence Larsen notes that ji-parameterized bisimilarity "lacks many of the properties presented in this paper". Concretely he mentions three properties.

First, that "$\sim_e$ is strictly included in $\sim_{\&e}$ for all environments e" (in general is meant [we use our notation for $\sim_{\&e}$ here]), corresponding to Proposition 3.7, (i) and (ii)). Second, that "thus the modal characterization for $\sim_e$ does not hold for $\sim_{\&e}$, and no other modal characterization seems immediate." This assessment stimulates us to work out (W1).

Finally third, Larsen writes that: "More important though is that the simulation ordering does not characterize the discrimination ordering generated by this alternative parameterized version[.]", in contrast with his impressive and surprising main result in [12], Theorem 2.5 here, which shows that that is the case for parameterized bisimilarity $\sim_e$. For this observation Larsen uses a counterexample that is slightly different from the one we use for Proposition 3.12, the corresponding statement here. Ji-parameterized bisimilarity fits nicely in the recently proposed framework [1, 2]: the authors thereof advocate to make a clear distinction between processes and what they call tests. The composition of processes with tests give rise to instrumentations, which are to be understood as "compiled binaries ready to be executed" [2, page 6]. Environments in our setting can be seen as tests, and hence join is a notion of composition. In contrast, $\sim_e$ lacks the composition operation (but still distinguishes processes and tests/environments). Below we formulate the question of a characterization of the discrimination preorder induced by ji-parameterized bisimilarity as the open problem (P1), and a specialization of this question as the open problem (P2).

Literature on parameterized versions of bisimilarity. Parameterized bisimilarity proved to be a very fruitful concept since its inception by Larsen in [11, 12]. His definition has been applied, specialized, and adapted in multiple ways in the meantime. Please see below for a few examples. But to the best of our knowledge this does not hold for the ji-parameterized concepts of simulatability and bisimilarity, apart from the passages in [12] that we cited and described above.

Parameterized bisimilarity in Larsen's definition [11, 12] has later been called 'relative bisimilarity' and 'relativized bisimilarity' in [14] by Larsen and Milner, who used it also for the practical purpose of verifying the Alternating Bit Protocol [14]. As pointed out in [7], it was also the basis for 'modal transition systems' to which a large body of work has been devoted since, see for example [16, 15, 3, 9].

Environment parameterized bisimulations in the sense of Larsens' definition or adapted and specialized variants of it have been used frequently, for example in [14, 13]. [19] introduces a notion of equivalence parameterized with respect to typing information, which, quoting from [19]: "can be seen as a disciplined instance of Larsen's, in which one uses types to express constraints on the behaviors of the observers, rather than explicitly writing all their possible behavior".

Current Work. We investigate modal-logical characterizations of (ji-)parameterized bisimilarity and simulatability, and of refinements of the modal characterizations already obtained by Larsen, and here.

(W1) We are working out a modal-logical characterization for ji-parameterized bisimilarity $\sim_{\&e}$ that is based on a game characterization of $\sim_{\&e}$. However, our characterization will not just be of a simple form comparable to Theorem 2.10 and Theorem 4.2, for all environments e:

$$p \sim_{\&e} q \iff \mathscr{M}(p) \cap \mathscr{F}(e) = \mathscr{M}(q) \cap \mathscr{F}(e) \quad \text{(for all image-finite processes } p, q\text{)}.$$

where $\mathscr{F} \subseteq \mathscr{M}$ would be appropriately defined formulas with then $\mathscr{F}(g) := \{\phi \in \mathscr{F} \mid g \vDash \phi\}$ defined for all environments g. It is nevertheless interesting to note that since $\nsim_{\&e} \subseteq \nsim_e$ holds (due to $\sim_e \subseteq \sim_{\&e}$ by Proposition 3.7, (i)), that whenever $p \nsim_{\&e} q$ holds, always $p \nsim_e q$ follows, and a formula $\phi \in (\mathscr{M}(p) \cap \overline{\mathscr{L}(e)}) \,\Delta\, (\mathscr{M}(q) \cap \overline{\mathscr{L}(e)})$ (where Δ denotes symmetric difference) that distinguishes p and q can always be found via Larsen's characterization, Theorem 2.10.

(W2) The restriction to image-finite processes for the modal characterizations of simulatability $\leq$ and bisimilarity $\sim$ by Hennessy and Milner (Theorem 2.8) can be dropped by permitting infinitary formulas with infinite conjunctions. Indeed, Milner has described such an adaptation for infinitary formulas in [18].

We want to obtain similar extensions to not necessarily image-finite processes for Larsen's characterization of $\sim_e$ (Theorem 2.10) and our ones of $\leq_e = \leq_{\&e}$ and $(\leq\geq)_e = (\leq\geq)_{\&e}$ (Theorem 4.2).

Open problems. As problems to which (satisfactory) answers have evaded us so far, we want to mention:

(P1) How can the discrimination order for $\sim_{\&e}$ be characterized? Note that a similar characterization in terms of simulatability $\leq$ as for the discrimination order of $\sim_e$ in Thm. 2.5 by Larsen, and for $\leq_{\&e}$ and $(\leq\geq)_{\&e}$ in Theorem. 3.14, is not possible due to Proposition 3.12.

(P2) Does equality of ji-parameterized bisimilarity with respect to environments e and f coincide with bisimilarity of e and f? Equivalently, does the implication "$\Leftarrow$" hold in the following statement (of which "$\Rightarrow$" is easy to verify), for all environments e and f:

$$e \sim f \quad \overset{?}{\Longleftrightarrow} \quad \sim_{\&e} = \sim_{\&f} .$$

Future research. As two lines of research for which the concept of ji-parameterized bisimilarity may lead to new insights we mention: a continuation of Larsen's work in his thesis [11] towards flexible formal methods for showing compositionality of program correctness (see (F1)), and consequences for finding interesting contextual behavioural metrics as introduced in [6] (see (F2)):

(F1) An interesting future work is the study of compositionality properties of $\sim_{\&e}$, that is the behavior of $\sim_{\&e}$ up to context. A context is typically defined as a syntactic process C (expressed in some process algebra) with a hole []. Notation $C[p]$ is used for the process obtained upon substitution of p for the hole in C. In general, $\sim_{\&e}$ is not preserved by contexts: Consider processes $p = a + b$, $q = a$, environment $e = a.b$ and context $C = a.[]$. We have that $p \sim_{\&e} q$, but $C[p] = a.a + b \not\sim_{\&e} a.a = C[q]$. Notice that the above example also applies to $\sim_e$. Indeed, including a process in a context intuitively also affects the environment, as shown in a study of the compositionality of $\sim_e$ in [12]. The idea in that work is to introduce parametric environment-transformer[4] T_C which preserves $\sim_e$ in the following sense:

$$p \sim_{T_C(e)} q \implies <C,p> \equiv_e <C,q> ,$$

where $<C,p> \equiv_e <C,q>$ intuitively means that "$C[p] \sim_{\&e} C[q]$ with C interacting identically with p and q" [12] (we omit the formal definition for brevity). We speculate that, for $\sim_{\&e}$, the requirement "C interacting identically with p and q" could be removed. If so, the compositionality of $\sim_{\&e}$ could be expressed as follows (for an appropriate environment- transformer T'_C):

$$p \sim_{\&T'_C(e)} q \implies C[p] \sim_{\&e} C[q] .$$

[4] We use a different notation than [12]. There, $T_C(e)$ is rendered as $wie_{\mathbb{E}}(C,e)$

(F2) The relatively recent work [6] shows that from $\sim_e$ (and quantitative generalizations of it) one can extract a generalized pseudo-metric between processes, where the codomain of the metric is the set of environments (under some closure assumptions). The idea is that the distance $d(p,q)$ between processes p,q is defined as the largest environment e (according to (2.1)) such that $p \sim_e q$. An obvious future work is exploring whether a metric can be extracted for $\sim_{\&e}$. The main challenge is finding the right notion of "largest environment" for $\sim_{\&e}$, which is related to open problem (P1).

Acknowledgment. We thank the reviewers for their questions about the literature on parameterized bisimilarity since its inception, and concerning our work on the logical characterization of $\sim_{\&e}$. We are also grateful for lists of comments that pointed us to necessary corrections and adaptations of details. We thank Clément Aubert for a direct discussion after the workshop, and in particular for pointing us to interesting connections with work [1, 2] of his and Daniele Varacca.

References

[1] Clément Aubert & Daniele Varacca (2021): *Process, Systems and Tests: Three Layers in Concurrent Computation*. In: *ICE*, *EPTCS* 347, pp. 1–21, doi:10.4204/EPTCS.347.1.

[2] Clément Aubert & Daniele Varacca (2022): *Processes against tests: On defining contextual equivalences*. *J. Log. Algebraic Methods Program.* 129, p. 100799, doi:10.1016/J.JLAMP.2022.100799.

[3] Nikola Beneš, Jan Křetínský, Kim G. Larsen, Mikael H. Møller & Jiří Srba (2011): *Parametric Modal Transition Systems*. In Tevfik Bultan & Pao-Ann Hsiung, editors: *Automated Technology for Verification and Analysis*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 275–289, doi:10.1007/978-3-642-24372-1_20.

[4] J. A. Bergstra & J. W. Klop (1990): *An Introduction to Process Algebra*. In J. C. M. Baeten, editor: *Applications of Process Algebra*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, p. 1–22, doi:10.1017/CBO9780511608841. Available at https://dspace.library.uu.nl/bitstream/handle/1874/13555/bergstra_90_introduction_process.pdf?sequence=3.

[5] Jan A Bergstra & Jan Willem Klop (1986): *Algebra of Communicating Processes*. Technical Report 89-138, CWI Amsterdam. Available at https://ir.cwi.nl/pub/1778/1778D.pdf.

[6] Ugo Dal Lago & Maurizio Murgia (2023): *Contextual Behavioural Metrics*. In Guillermo A. Pérez & Jean-François Raskin, editors: *34th International Conference on Concurrency Theory (CONCUR 2023)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 279, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 38:1–38:17, doi:10.4230/LIPIcs.CONCUR.2023.38.

[7] Uli Fahrenberg, Kim Guldstrand Larsen, Axel Legay & Louis-Marie Traonouez (2014): *Parametric and Quantitative Extensions of Modal Transition Systems*. In Saddek Bensalem, Yassine Lakhneck & Axel Legay, editors: *From Programs to Systems. The Systems perspective in Computing*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 84–97, doi:10.1007/978-3-642-54848-2_6.

[8] Matthew Hennessy & Robin Milner (1985): *Algebraic Laws for Nondeterminism and Concurrency*. *J. ACM* 32(1), p. 137–161, doi:10.1145/2455.2460.

[9] Michael Huth, Radha Jagadeesan & David Schmidt (2001): *Modal Transition Systems: A Foundation for Three-Valued Program Analysis*. In David Sands, editor: *Programming Languages and Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 155–169, doi:10.1007/3-540-45309-1_11.

[10] Ugo Dal Lago & Maurizio Murgia (2023): *Contextual Behavioural Metrics (Extended Version)*, doi:10.48550/arXiv.2307.07400. arXiv:2307.07400.

[11] Kim G. Larsen (1986): *Context-Dependent Bisimulation between Processes*. Ph.D. thesis, University of Edinburgh. Available at https://era.ed.ac.uk/bitstream/handle/1842/11030/Larsen1986.pdf.

[12] Kim G. Larsen (1987): *A Context Dependent Equivalence between Processes*. *Theoretical Computer Science* 49(2), pp. 185–215, doi:10.1016/0304-3975(87)90007-7.

[13] Kim G. Larsen, Ulrik Larsen & Andrzej Wasowski (2005): *Color-Blind Specifications for Transformations of Reactive Synchronous Programs*. In Maura Cerioli, editor: *Fundamental Approaches to Software Engineering*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 160–174, doi:10.1007/978-3-540-31984-9_13.

[14] Kim G. Larsen & Robin Milner (1992): *A Compositional Protocol Verification Using Relativized Bisimulation*. *Information and Computation* 99(1), pp. 80–108, doi:10.1016/0890-5401(92)90025-B.

[15] Kim G. Larsen, Ulrik Nyman & Andrzej Wąsowski (2007): *On Modal Refinement and Consistency*. In Luís Caires & Vasco T. Vasconcelos, editors: *CONCUR 2007 – Concurrency Theory*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 105–119, doi:10.1007/978-3-540-74407-8_8.

[16] Kim Guldstrand Larsen (1990): *Modal Specifications*. In Joseph Sifakis, editor: *Automatic Verification Methods for Finite State Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 232–246, doi:10.1007/3-540-52148-8_19.

[17] Robin Milner (1980): *A Calculus of Communicating Systems*, 1980 edition. Springer Berlin Heidelberg, doi:10.1007/3-540-10235-3.

[18] Robin Milner (1985): *Lectures on a Calculus for Communicating Systems*. In Stephen D. Brookes, Andrew William Roscoe & Glynn Winskel, editors: *Seminar on Concurrency*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 197–220, doi:10.1007/3-540-15670-4_10.

[19] Benjamin C. Pierce & Davide Sangiorgi (2000): *Behavioral Equivalence in the Polymorphic Pi-Calculus*. *J. ACM* 47(3), p. 531–584, doi:10.1145/337244.337261.

[20] Davide Sangiorgi (2011): *Introduction to Bisimulation and Coinduction*. Cambridge University Press, doi:10.1017/CBO9780511777110.

A Formalization of the Reversible Concurrent Calculus CCSK^{P} in Beluga

Gabriele Cecilia
School of Computer & Cyber Sciences,
Augusta University, Augusta, USA

gcecilia@augusta.edu

Reversible concurrent calculi are abstract models for concurrent systems in which any action can potentially be undone. Over the last few decades, different formalisms have been developed and their mathematical properties have been explored; however, none have been machine-checked within a proof assistant. This paper presents the first Beluga formalization of the Calculus of Communicating Systems with Keys and Proof labels (CCSK^{P}), a reversible extension of CCS. Beyond the syntax and semantics of the calculus, the encoding covers state-of-the-art results regarding three relations over proof labels – namely, dependence, independence and connectivity – which offer new insights into the notions of causality and concurrency of events. As is often the case with formalizations, our encoding introduces adjustments to the informal proof and makes explicit details which were previously only sketched, some of which reveal to be less straightforward than initially assumed. We believe this work lays the foundations for future reversible concurrent calculi formalizations.

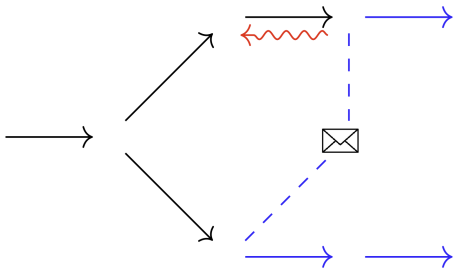
1 Introduction

Concurrency in computer science refers to the simultaneous execution of multiple operations or computations in a shared environment. It is a fundamental aspect of modern computing, with practical use in several domains such as operating systems, networking and distributed systems. Process calculi like CCS [16] and the π-calculus [17] are well-studied and established mathematical models for formally describing and reasoning about concurrent systems.

In recent years, reversing computations in concurrent systems has gained significant attention, with applications in fields like hardware, software and biochemistry [22]. Enriching concurrent systems with reversibility poses its own set of challenges: for instance, it requires providing some kind of history-preserving mechanism to take track of past actions. Additionally, undoing computation steps in a parallel setting is more complex than in a sequential system: as explained in Fig. 1, reversing a specific action performed by a single thread may require knowing, and eventually undoing, the actions of the other threads it has previously interacted with.

Reversible concurrent calculi address such challenges in various ways. For example, Reversible CCS (RCCS) [11] equips processes with a memory that records information about past computations; conversely, CCS with Keys (CCSK) [20] associates unique keys to each forward action. The latter has been recently upgraded to CCSK with Proof labels (CCSK^{P}) [4], which features a proved transition system in the fashion of Degano and Priami [12]; proof labels enable the definition of dependence and independence for both forward and backward transitions. In this framework, the contributions brought by Aubert et al. [3] merit attention. The authors are the first to introduce separate axioms for the relations of dependence, independence and connectivity on proof labels: such relations are proved to be sound, interrelated, and linked to the broader notions of concurrency and causality of events. Additionally, the

C. Aubert, C. Di Giusto, S. Fowler & V. K. I Pun (Eds.):
18th Interaction and Concurrency Experience (ICE 2025)
EPTCS 425, 2025, pp. 55–72, doi:10.4204/EPTCS.425.5



The diagram on the left illustrates the duplication of a thread, followed by a message exchange between the resulting branches and some further independent computation steps. To undo the computation step marked in red, which belongs to one of the two threads and precedes their interaction, it is necessary to first undo each of the steps marked in blue, including those of the other thread.

Figure 1: Example of reversal of computation steps in a concurrent setting.

authors outline the difference between various kinds of bisimulations, such as the history preserving bisimulation for CCS.

Formal verification has become a cornerstone in the development of new systems, certifying the correctness of their syntax, semantics and behavioral properties in the most reliable way. In the case of concurrent calculi, there is a long tradition which spans from the early mechanizations by [18] and [15] in HOL to the recent contributions of the Concurrent Calculi Formalisation Benchmark [6]; we also recall the Rocq formalization of the π-calculus by Honsell et al. [14], which has been the baseline for numerous higher-order abstract syntax (HOAS) [19] mechanizations. When it comes to reversible concurrent calculi, however, the landscape looks rather different. Despite the availability of C# and Java implementations of CCSK [10][1], no machine-checked formalization of reversible concurrent calculi currently exists – at least to the best of our knowledge.

This paper presents the first formalization of CCSK^P in Beluga [21]. Our encoding covers the core definitions of the system: its syntax, semantics, and the relations of dependence, independence and connectivity on proof labels. Additionally, this work formalizes the central results presented in Sections 3 and 4 of [3], including the complementarity of dependence and independence and the relationship between connectivity of transitions and proof labels. The proofs come along with a library of auxiliary lemmas regarding processes, keys and transitions.

Formalizations typically require small adjustments to fit the proof assistant's framework, while carefully addressing any of the details which are taken for granted in the informal proof. This encoding is no exception: the formalization process led to minor refinements in the definition of connectivity over proof labels and clarified the proof of one of the aforementioned results, which called for a different approach separating base and inductive cases. Beluga, as a proof assistant, is ideal for reasoning about deductive systems together with their meta-theory, as it naturally supports encodings of object-level binding constructs through higher-order abstract syntax and allows pairing terms with the contexts that give them meaning [21]. Although our HOAS encoding leverages Beluga's strengths and showcases its versatility, it also deals with limitations such as the lack of syntactic sugar for existentials or conjunctions; considerations on its adoption are further elaborated in the conclusions.

The paper is structured as follows. Section 2 provides an informal description of CCSK^P and the results under our study. Section 3 presents the Beluga formalization of such notions and properties. Section 4 contains a technical overview of the formalization, a summary of our contributions and possible future work directions. The artifact is archived in Zenodo [7] and available in the associated GitHub repository `https://github.com/CinRC/A-Beluga-Formalization-of-CCSKP`.

2 CCSK$^{\mathrm{P}}$

In this section we recall the main definitions and properties of CCSK$^{\mathrm{P}}$, as outlined in [3]. We assume familiarity with the basic notions of CCS. Definitions are hyperlinked to their encoding in the repository.

2.1 Syntax

As in the standard CCS, we assume the existence of an infinite set N of *names*, ranged over by a, b, c, with a bijection $\bar{\cdot}\colon \mathsf{N} \to \overline{\mathsf{N}}$ denoting the *complement* of a name; names and complementary names respectively denote input and output ports for processes. We define the set of *labels* L as $\mathsf{N} \cup \overline{\mathsf{N}} \cup \{\tau\}$, where τ denotes the interaction of concurrent processes. L is ranged over by α, while $\mathsf{L} \setminus \{\tau\}$ is ranged over by λ.

To introduce reversibility, CCSK extends the syntax of CCS with a denumerable set K of *keys*, ranged over by k, m, n. Labels are paired with keys to define *keyed labels*, which are elements of the cartesian product $\mathsf{L} \times \mathsf{K}$ and are represented as $a[k], b[m]$; the set of keyed labels is also denoted as L_{K}.

Processes are defined as in the ordinary CCS, with the addition of keyed prefixes and without operators for recursion or replication:

$$\begin{array}{rcllll} X, Y & ::= & \mathbf{0} & \text{(Inactive)} & \mid \alpha.X & \text{(Prefix)} \\ & & \mid \alpha[k].X & \text{(Keyed prefix)} & \mid X + Y & \text{(Sum)} \\ & & \mid X \mid Y & \text{(Parallel composition)} & \mid X \setminus a & \text{(Restriction)} \end{array}$$

The set of processes is denoted as $\mathbb{X}$. When preceded by a (keyed) prefix, the inactive process $\mathbf{0}$ is usually omitted; the binding power of the operators, from highest to lowest, is $\setminus a, \alpha[k], \alpha, \mid$ and $+$. In restrictions $X \setminus a$, the occurrences of the name a in X are said to be bound; all other occurrences of names and keys in processes are considered free. Processes that are α-equivalent, i.e., that differ only in the choice of their bound names, will be identified. Unlike [3], restrictions only bind names and not complementary names: this choice does not rule out any significant process (since $X \setminus a$ or $X \setminus \bar{a}$ have the same behaviour) and leads to a clearer correspondence between processes and their encoding.

The set of keys occurring in a process is denoted as $\mathsf{keys}(X)$. A process for which $\mathsf{keys}(X)$ is empty is said to be *standard*: in this case, we write that $\mathsf{std}(X)$ holds.

2.2 Semantics

The key feature of CCSK$^{\mathrm{P}}$ is the notion of *proof keyed labels*:

$$\theta \ ::= v\alpha[k] \quad \mid \quad v\langle |_{\mathrm{L}} v_1 \lambda[k], |_{\mathrm{R}} v_2 \overline{\lambda}[k] \rangle$$

where v, v_1 and v_2 range over strings of symbols $\{|_{\mathrm{L}}, |_{\mathrm{R}}, +_{\mathrm{L}}, +_{\mathrm{R}}\}$. We denote the set of proof keyed labels as $\mathsf{L}^{\mathsf{P}}_{\mathsf{K}}$, and refer to its elements simply as "proof labels" for brevity. The following functions ℓ and $\not{k}$ map each proof label to its underlying label and key, respectively:

$$\ell(v\alpha[k]) = \alpha \qquad \ell(v\langle |_{\mathrm{L}} v_1 \lambda[k], |_{\mathrm{R}} v_2 \overline{\lambda}[k] \rangle) = \tau \qquad \not{k}(v\alpha[k]) = k \qquad \not{k}(v\langle |_{\mathrm{L}} v_1 \lambda[k], |_{\mathrm{R}} v_2 \overline{\lambda}[k] \rangle) = k$$

Semantics is given by the *labelled transition system* $(\mathbb{X}, \mathsf{L}^{\mathsf{P}}_{\mathsf{K}}, \overset{\theta}{\mapsto})$, where $\overset{\theta}{\mapsto}$ denotes the union of the forward and backward transitions displayed in Fig. 2. We will refer to the union of forward and backward transitions as *combined* transitions. Given a transition $X \overset{\theta}{\mapsto} Y$, the process X is said to be its *source*, while Y is said to be its *target*.

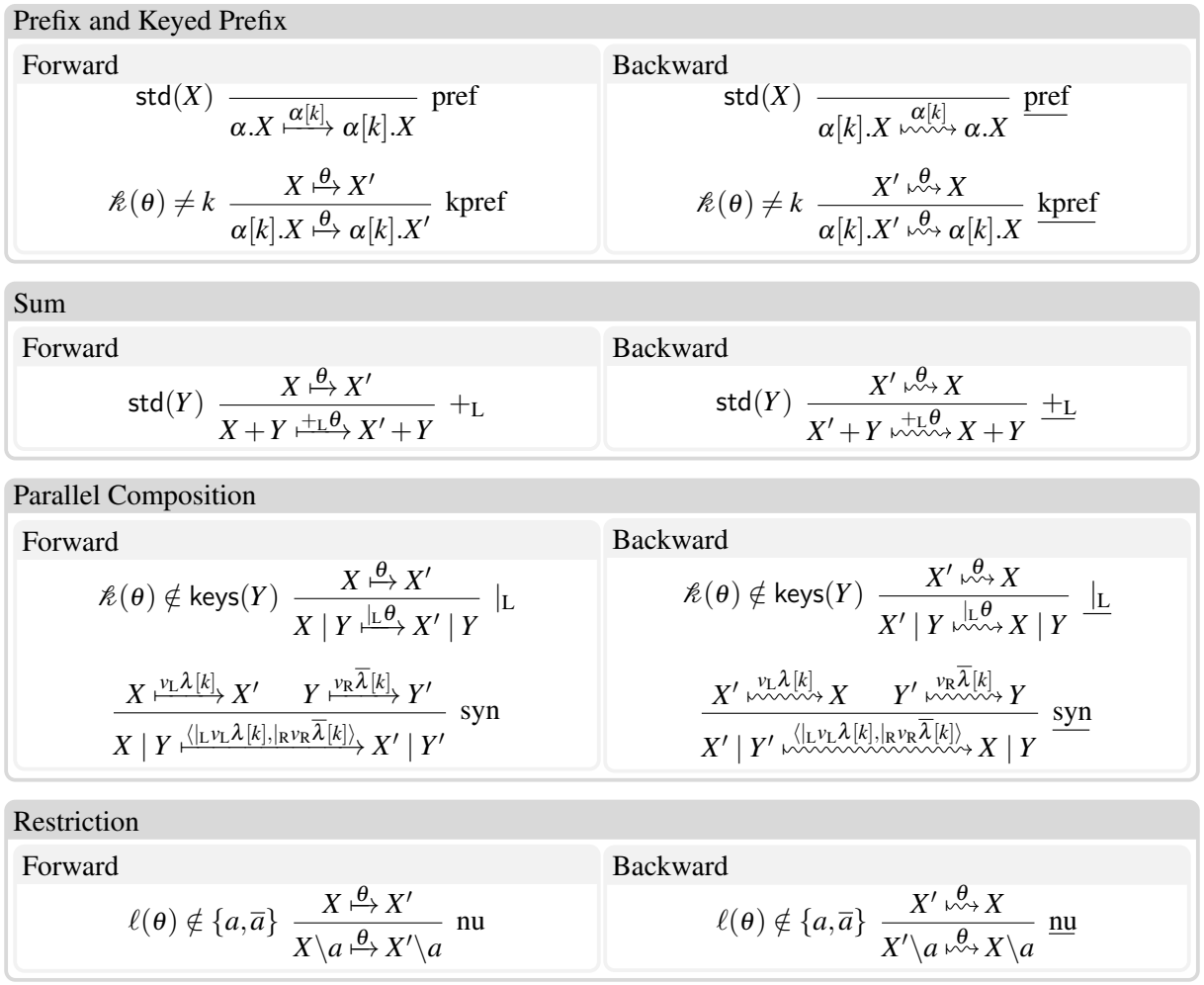


Figure 2: Forward and backward transition rules for CCSK^P (right rules for $\mid$ and $+$ omitted).

Example 1 Consider a webpage that allows the user to interact via two independent buttons: one to toggle between light and dark mode, and another to switch between two different languages. The initial state of the system can be modeled as the parallel composition $m \mid l$, where the labels m and l represent the actions to switch the visual mode and the language, respectively.

The action of changing the visual mode can be represented by the following forward transition: $m \mid l \xmapsto{|_L m[k]} m[k] \mid l$. The target process preserves the label m and pairs it with a fresh key k. The proof label $|_L m[k]$ not only stores the label m and key k used in the transition, but also indicates that the action occurred on the left-hand side of a parallel composition.

Suppose the user now wishes to revert to the previous visual mode: pressing the mode button again can be interpreted as undoing the previously executed action. This can be modeled by the following backward transition, which removes the key associated with the earlier forward step: $m[k] \mid l \xmapsto{|_L m[k]} m \mid l$.

Two transitions are said to be *composable* if they can be performed consecutively – that is, the target of the first transition is the source of the second transition. A *path* is a (potentially empty) sequence of composable transitions and can be denoted as $X \mapsto^* Y$, where X is the source of the first transition (also called the *source* of the path) and Y is the target of the last transition (also called the *target* of the path); in

other words, $\mapsto\!\!\!\!\rightarrow^*$ is the reflexive and transitive closure of $\mapsto\!\!\!\!\rightarrow$. A process X is *reachable* if there exists a path whose target is X and whose source is a standard process. This process, which can be proved to be unique (cf. Lemma B.13 in [2]), is called the *origin* of X and is denoted as O_X.

Reachability allows to rule out faulty processes which are syntactically well-formed, but whose particular selection of keys is inconsistent. For instance, this arises when the same key denotes successive actions, as in the process $a[k].b[k]$, or when keys internally form a cycle, as in the deadlocked process $a[k].b[m] \mid \bar{b}[m].\bar{a}[k]$, where neither action can be undone because of the presence of its associated key in the other thread. From this point on, each process will be assumed to be reachable.

The *loop lemma* (cf. Lemma 3.8 in [3]) is an important result characterizing reversible labelled transition systems. It states that any transition $X \overset{\theta}{\mapsto\!\!\!\!\rightarrow} Y$ can be reversed, yielding a transition $Y \overset{\theta}{\mapsto\!\!\!\!\rightarrow} X$; moreover, the reversing operator is an involution (i.e., reversing a transition twice returns the original transition). The validity of the loop lemma follows directly from the symmetry of the LTS (Labelled Transition System) rules presented in Fig. 2.

Connectivity Relation

Action

$$\frac{}{\alpha[k] \curlyvee \theta}\ \mathrm{A}^1 \qquad \frac{}{\theta \curlyvee \alpha[k]}\ \mathrm{A}^2$$

Choice

$$\frac{\theta_1 \curlyvee \theta_2}{+_{\mathrm{d}}\theta_1 \curlyvee +_{\mathrm{d}}\theta_2}\ \mathrm{C}^1_{\mathrm{d}} \qquad \frac{}{+_{\mathrm{d}}\theta_1 \curlyvee +_{\bar{\mathrm{d}}}\theta_2}\ \mathrm{C}^2_{\mathrm{d}}$$

Parallel

$$\frac{\theta_1 \curlyvee \theta_2}{|_{\mathrm{d}}\theta_1 \curlyvee |_{\mathrm{d}}\theta_2}\ \mathrm{P}^1_{\mathrm{d}} \qquad \frac{}{|_{\mathrm{d}}\theta_1 \curlyvee |_{\bar{\mathrm{d}}}\theta_2}\ \mathrm{P}^2_{\mathrm{d}}$$

Synchronization

$$\frac{\theta \curlyvee \theta_{\mathrm{d}}}{|_{\mathrm{d}}\theta \curlyvee \langle |_{\mathrm{L}}\theta_{\mathrm{L}}, |_{\mathrm{R}}\theta_{\mathrm{R}}\rangle}\ \mathrm{S}^1_{\mathrm{d}} \qquad \frac{\theta_{\mathrm{d}} \curlyvee \theta}{\langle |_{\mathrm{L}}\theta_{\mathrm{L}}, |_{\mathrm{R}}\theta_{\mathrm{R}}\rangle \curlyvee |_{\mathrm{d}}\theta}\ \mathrm{S}^2_{\mathrm{d}}$$

$$\frac{\theta_1 \curlyvee \theta_1' \qquad \theta_2 \curlyvee \theta_2'}{\langle |_{\mathrm{L}}\theta_1, |_{\mathrm{R}}\theta_2\rangle \curlyvee \langle |_{\mathrm{L}}\theta_1', |_{\mathrm{R}}\theta_2'\rangle}\ \mathrm{S}^3$$

Dependence Relation

Action

$$\frac{}{\alpha[k] \asymp \theta}\ \mathrm{A}^1 \qquad \frac{}{\theta \asymp \alpha[k]}\ \mathrm{A}^2$$

Choice

$$\frac{\theta \asymp \theta'}{+_{\mathrm{d}}\theta \asymp +_{\mathrm{d}}\theta'}\ \mathrm{C}^1_{\mathrm{d}} \qquad \frac{}{+_{\mathrm{d}}\theta \asymp +_{\bar{\mathrm{d}}}\theta'}\ \mathrm{C}^2_{\mathrm{d}}$$

Parallel

$$\frac{\theta \asymp \theta'}{|_{\mathrm{d}}\theta \asymp |_{\mathrm{d}}\theta'}\ \mathrm{P}^1_{\mathrm{d}} \qquad \frac{\not{k}(\theta) = \not{k}(\theta')}{|_{\mathrm{d}}\theta \asymp |_{\bar{\mathrm{d}}}\theta'}\ \mathrm{P}^2_{\mathrm{d}}$$

Synchronization

$$\frac{\theta \asymp \theta_{\mathrm{d}}}{|_{\mathrm{d}}\theta \asymp \langle |_{\mathrm{L}}\theta_{\mathrm{L}}, |_{\mathrm{R}}\theta_{\mathrm{R}}\rangle}\ \mathrm{S}^1_{\mathrm{d}} \qquad \frac{\theta_{\mathrm{d}} \asymp \theta}{\langle |_{\mathrm{L}}\theta_{\mathrm{L}}, |_{\mathrm{R}}\theta_{\mathrm{R}}\rangle \asymp |_{\mathrm{d}}\theta}\ \mathrm{S}^2_{\mathrm{d}}$$

$$\frac{\theta_i \asymp \theta_i' \qquad \theta_j \curlyvee \theta_j' \qquad i,j \in \{1,2\}, i \neq j}{\langle |_{\mathrm{L}}\theta_1, |_{\mathrm{R}}\theta_2\rangle \asymp \langle |_{\mathrm{L}}\theta_1', |_{\mathrm{R}}\theta_2'\rangle}\ \mathrm{S}^3$$

Independence Relation

Action

(empty)

Choice

$$\frac{\theta \between \theta'}{+_{\mathrm{d}}\theta \between +_{\mathrm{d}}\theta'}\ \mathrm{C}^1_{\mathrm{d}}$$

Parallel

$$\frac{\theta \between \theta'}{|_{\mathrm{d}}\theta \between |_{\mathrm{d}}\theta'}\ \mathrm{P}^1_{\mathrm{d}} \qquad \frac{\not{k}(\theta) \neq \not{k}(\theta')}{|_{\mathrm{d}}\theta \between |_{\bar{\mathrm{d}}}\theta'}\ \mathrm{P}^2_{\mathrm{d}}$$

Synchronization

$$\frac{\theta \between \theta_{\mathrm{d}}}{|_{\mathrm{d}}\theta \between \langle |_{\mathrm{L}}\theta_{\mathrm{L}}, |_{\mathrm{R}}\theta_{\mathrm{R}}\rangle}\ \mathrm{S}^1_{\mathrm{d}} \qquad \frac{\theta_{\mathrm{d}} \between \theta}{\langle |_{\mathrm{L}}\theta_{\mathrm{L}}, |_{\mathrm{R}}\theta_{\mathrm{R}}\rangle \between |_{\mathrm{d}}\theta}\ \mathrm{S}^2_{\mathrm{d}}$$

$$\frac{\theta_1 \between \theta_1' \qquad \theta_2 \between \theta_2'}{\langle |_{\mathrm{L}}\theta_1, |_{\mathrm{R}}\theta_2\rangle \between \langle |_{\mathrm{L}}\theta_1', |_{\mathrm{R}}\theta_2'\rangle}\ \mathrm{S}^3$$

Figure 3: Causality relations on proof labels.

Finally, the binary relations of *connectivity*, *dependence* and *independence* on proof labels, respectively denoted as $\curlyvee, \asymp$ and ι, are defined by the rules displayed in Fig. 3, where the label d ranges over $\{\mathrm{L}, \mathrm{R}\}$ and $\overline{\mathrm{d}}$ denotes the opposite of d (i.e., $\overline{\mathrm{L}} = \mathrm{R}$ and $\overline{\mathrm{R}} = \mathrm{L}$). Such relations will be referred to as *causality relations* for brevity. Compared to [3], the rule A^2 for connectivity and dependence has been slightly modified, ensuring that each relation is symmetric and simplifying their encoding. This comes at the cost of losing uniqueness in derivations of judgements such as $\theta_1 \curlyvee \theta_2$; however, this property has been shown not to be required for the purposes of our development.

Example 2 The process $m \mid l$, introduced in Example 1 to model a webpage, can perform a forward transition labelled by $|_L m[k_1]$, representing the toggling of the visual mode. It can also perform a transition $m \mid l \xmapsto{|_R l[k_2]\rightarrow} m \mid l[k_2]$, denoting the change of the language of the webpage. The two transitions are independent, as the order in which they are executed does not affect the resulting state. This is reflected in the independence of the two proof labels $|_L m[k_1]$ and $|_R l[k_2]$, which follows from the $\mathrm{P}^2_{\mathrm{L}}$ rule in Fig. 3.

Conversely, consider the process $a.b \mid \bar{b}$. It can perform a forward transition labelled by $|_L a[k]$, followed by another forward transition labelled by $|_L b[n]$. However, these transitions cannot be performed in reverse order, since the input action along b is only enabled after the input action along a has occurred; the two transitions are thus causally related. This is reflected in the dependence of the two proof labels $|_L a[k]$ and $|_L b[n]$, which follows from the $\mathrm{P}^1_{\mathrm{L}}$ and A^1 rules in Fig. 3.

2.3 Properties of causality relations

We now turn to the theorems and lemmas object of our study. Their complete proof can be found in [2], the technical report accompanying [3]. The following theorem specifies the relationship between connectivity of transitions and connectivity of proof labels:

Theorem 2.1 (cf. Proposition 4.4 in [3])

(i) If $t_1 : X_1 \overset{\theta_1}{\longmapsto\!\!\!\!\rightarrow} X_1'$ and $t_2 : X_2 \overset{\theta_2}{\longmapsto\!\!\!\!\rightarrow} X_2'$ are connected, then $\theta_1 \curlyvee \theta_2$. ↗

(ii) If $\theta_1 \curlyvee \theta_2$, then there exist $t_1 : X_1 \overset{\theta_1}{\longmapsto\!\!\!\!\rightarrow} X_1'$ and $t_2 : X_2 \overset{\theta_2}{\longmapsto\!\!\!\!\rightarrow} X_2'$ such that t_1 and t_2 are connected. ↗

The proof of Theorem 2.1(i) relies on the fact that $O_{X_1} = O_{X_2}$ and proceeds by induction over such origin process: recall that each process is assumed to be reachable and, therefore, has an origin. The equality of O_{X_1} and O_{X_2} follows from the two lemmas:

Lemma 2.2 *For all reachable processes X and Y, there exists a path $X \mapsto\!\!\!\!\rightarrow^* Y$ iff $O_X = O_Y$.*

Lemma 2.3 *If $t_1 : X_1 \overset{\theta_1}{\longmapsto\!\!\!\!\rightarrow} X_1'$ and $t_2 : X_2 \overset{\theta_2}{\longmapsto\!\!\!\!\rightarrow} X_2'$ are connected, then $O_{X_1} = O_{X_2}$.*

Conversely, the proof of Theorem 2.1(ii) proceeds by structural induction over the given hypothesis $\theta_1 \curlyvee \theta_2$ and relies on the following:

Definition 2.4 (Realisation) *A process X* realises the proof label θ *if there exist X_1 and X_2 such that $X \mapsto\!\!\!\!\rightarrow^* X_1 \overset{\theta}{\longmapsto\!\!\!\!\rightarrow} X_2$.* ↗

Lemma 2.5 *For every proof label θ, there exists a process that realises it, and we denote it $r(\theta)$.* ↗

Next, the following theorem states the complementarity of the dependence and independence relations:

Theorem 2.6 (cf. Theorem 4.9 in [3])

For all θ_1, θ_2,

(i) If $\theta_1 \iota \theta_2$ then $\theta_1 \curlyvee \theta_2$. ↗

(ii) If $\theta_1 \asymp \theta_2$ then $\theta_1 \curlyvee \theta_2$. ↗

(iii) If $\theta_1 \curlyvee \theta_2$ then either $\theta_1 \iota \theta_2$ or $\theta_1 \asymp \theta_2$, but not both. ↗

This theorem is proved by induction over the structure of the given binary relation.

3 Beluga Formalization

In this section we outline the key points of the Beluga formalization of the notions presented in Section 2. Definitions and proofs omitted for brevity are hyperlinked to their encoding in the repository.

3.1 Syntax

Beluga is structured in two layers: the LF (Logical Frameworks [13]) level, which is used to specify the formal system under study, and the computation level, which supports programming with LF data [21]. To encode the syntax of our system, only the former level is deployed. Names, keys, labels and processes are encoded using the LF types displayed in Fig. 4.

```
LF names: type =;
LF keys: type =
  | z: keys
  | s: keys → keys;
LF labels: type =
  | inp: names → labels
  | out: names → labels
  | tau: labels;

LF proc: type =
  | null: proc                                  % 0
  | pref: labels → proc → proc                  % A.X
  | kpref: labels → keys → proc → proc          % A[k].X
  | sum: proc → proc → proc                     % X+Y
  | par: proc → proc → proc                     % X|Y
  | nu: (names → proc) → proc;                  % X\a
```

Figure 4: Encoding of the syntax of CCSK$^{\mathrm{P}}$.

Since names in CCSK$^{\mathrm{P}}$ are an infinite set without any additional assumption, they are represented by a type `names` without constructors; as explained in [8], this type will be dynamically inhabited by variables introduced through contexts. This is enabled by the following line of code:

```
schema ctx = names;
```

This line declares contexts made of a finite collection of distinct variables of type `names`, identified via the keyword `ctx`. Thanks to this setup, we can work with *contextual processes* of the form `[g ⊢ X]`, i.e., processes `X` whose free names are drawn from the context `g`. Contextual objects live in the computation level.

Keys are by assumption denumerable, and the LTS rules for keyed prefixes require equality of keys to be decidable. Both conditions are satisfied by encoding keys explicitly as natural numbers.[1] An alternative approach would be to rely on contexts, as is done for names: however, this would require managing mixed contexts of names and keys, and having a more complex encoding of transitions and paths.

Restrictions $X \setminus a$ are represented by terms of the form `(nu \a.(X a))`, where `\x.(f x)` is Beluga's notation for functions `f` mapping `x` to `f(x)`: following the higher-order abstract syntax (HOAS) paradigm, the bound name a is represented as the implicit argument of a meta-language function `\a.(X a)` from `names` to `proc`. In this way, we leverage the meta-language implementation of binders to achieve α-renaming and capture-avoiding substitutions for free.

An important but often overlooked aspect of formalizations is the *adequacy* of the encoding: the encoding must constitute a faithful representation of the original system into study [9]. Adequacy is generally established by proving the existence of a compositional bijection between the mathematical model and its formalized counterpart. The discussion of the adequacy of our encoding is omitted due to space constraints.

[1] Note that properties such as decidability of equality must be stated and proved manually, as Beluga does not provide a built-in library of properties of natural numbers.

3.2 Semantics

Proof labels are encoded by the type pr_lab in Fig. 5. Rather than directly modeling the informal definition of proof labels, by defining strings over the symbols $\{|_L, |_R, +_L, +_R\}$ as lists, we are introducing four constructors (pr_suml, pr_sumr, etc.) that build proof labels incrementally by appending one symbol at a time. This provides a stronger induction principle and streamlines the encoding of LTS rules and subsequent proofs.

```
LF pr_lab: type =
  | pr_base: labels → keys → pr_lab
  | pr_suml: pr_lab → pr_lab
  | pr_sumr: pr_lab → pr_lab
  | pr_parl: pr_lab → pr_lab
  | pr_parr: pr_lab → pr_lab
  | pr_sync: pr_lab → pr_lab → pr_lab;
LF valid: pr_lab → type =
  | v_base: valid (pr_base A K)
  | v_suml: valid T → valid (pr_suml T)
  | v_sumr: valid T → valid (pr_sumr T)
  | v_parl: valid T → valid (pr_parl T)
  | v_parr: valid T → valid (pr_parr T)
  | v_synl: valid T1 → valid T2 → lab T1 (inp A) → key T1 K
        → lab T2 (out A) → key T2 K → valid (pr_sync T1 T2)
  | v_synr: valid T1 → valid T2 → lab T1 (out A) → key T1 K
        → lab T2 (inp A) → key T2 K → valid (pr_sync T1 T2);
LF fstep: proc → pr_lab → proc → type =
  | fs_pref: std X → fstep (pref A X) (pr_base A K) (kpref A K X)
  | fs_kpref: fstep X T X' → key T M → neq K M
         → fstep (kpref A K X) T (kpref A K X')
  | fs_suml: fstep X T X' → std Y → fstep (sum X Y) (pr_suml T) (sum X' Y)
  | fs_sumr: fstep Y T Y' → std X → fstep (sum X Y) (pr_sumr T) (sum X Y')
  | fs_parl: fstep X T X' → key T K → notin K Y
        → fstep (par X Y) (pr_parl T) (par X' Y)
  | fs_parr: fstep Y T Y' → key T K → notin K X
        → fstep (par X Y) (pr_parr T) (par X Y')
  | fs_synl: fstep X T1 X' → lab T1 (inp L) → key T1 K
        → fstep Y T2 Y' → lab T2 (out L) → key T2 K
        → fstep (par X Y) (pr_sync T1 T2) (par X' Y')
  | fs_synr: fstep X T1 X' → lab T1 (out L) → key T1 K
        → fstep Y T2 Y' → lab T2 (inp L) → key T2 K
        → fstep (par X Y) (pr_sync T1 T2) (par X' Y')
  | fs_nu: ({a:names} fstep (X a) T (X' a)) → fstep (nu X) T (nu X');
```

```
LF lab: pr_lab → labels → type =
  | lab_base: lab (pr_base A K) A
  | lab_suml: lab T A → lab (pr_suml T) A
  | lab_sumr: lab T A → lab (pr_sumr T) A
  | lab_parl: lab T A → lab (pr_parl T) A
  | lab_parr: lab T A → lab (pr_parr T) A
  | lab_sync: lab (pr_sync T1 T2) tau;
```

Figure 5: Encoding of the semantics of CCSK^P.

In Beluga, predicates are encoded as type families, i.e., types parametrized by arguments: a predicate $P(x_1,\ldots,x_n)$ holds iff the corresponding type (P x_1 … x_n) is not empty. Type families are also used to encode functions, identified with their graph, as in the case of the functions ℓ and $\not{k}$ returning the label and key of a proof label: the former is encoded by the type family lab in Fig. 5, while the latter is encoded by the type family key, here omitted for brevity. For example, given a proof label θ and a label α, represented as T and A in the encoding, the type lab T A is inhabited iff $\ell(\theta) = \alpha$.

Our encoding of proof labels pays the price of being over-expressive: the constructor `pr_sync` accepts any two proof labels regardless of their key or label, generating terms that fall outside the original definition. For example, the term `(pr_sync (pr_base A K) (pr_base B M))` has type `pr_lab` for any labels `A`, `B` and keys `K`, `M`, while its counterpart $\langle |_{\mathrm{L}}\alpha[k], |_{\mathrm{R}}\beta[m]\rangle$ is well-defined only if $\beta = \overline{\alpha}$ and $k = m$. While this is harmless in most of our development, since such spurious terms do not label any actual transition, it becomes an issue when proving theorems universally quantified on proof labels, such as Lemma 2.5. To address this problem, we introduce an additional predicate `valid`, displayed in Fig. 5, which filters out the spurious terms. It can be proved the existence of a bijection between proof labels and the set of elements `T` of type `pr_lab` for which `valid T` holds. From this point forward, we will refer to such terms as *valid* proof labels.

Forward and backward LTS rules are defined through the type families `fstep` and `bstep` in Fig. 5 (with the latter omitted here for brevity). These rules rely on the additional type families `std`, `notin` and `neq`, hyperlinked to their formalization in the repository, which respectively hold when a process is standard, when a key does not occur in a process, and when two keys are not equal. The parameters `X` and `X'` in the `fs_nu` rule are functions from `names` to `proc`, whose arguments represent the binders of the restrictions. The universal quantification `{a:names}` is used to abstract over the particular choice of the binder; moreover, a or $\bar{a}$ does not occur in the proof label `T`, since such parameter does not depend on `a` within the body of the universal quantification.

Combined transitions, paths, reachable processes and connected transitions are defined as follows:

```
LF step: proc → pr_lab → proc → type =
  | fw: fstep X T X' → step X T X'
  | bw: bstep X' T X → step X' T X;
LF step*: proc → proc → type =
  | id_s*: step* X X
  | st_s*: step X T Y → step* X Y
  | tr_s*: step* X Y → step* Y Z → step* X Z;
LF reachable: proc → type =
  | rch: std X → step* X Y → reachable Y;
LF conn_tr: step X T1 X' → step Y T2 Y' → type =
  | ct: {S1:step X T1 X'}{S2: step Y T2 Y'} step* X Y' → conn_tr S1 S2;
```

Paths, or multi-step transitions, can be encoded equivalently using only two constructors; in this development, the more verbose version has been adopted as it simplified the proof search.

Finally, the relations of connectivity, dependence and independence are encoded through three type families `conn`, `dep` and `indep`. The former is displayed in Fig. 6. While some of the rules in Fig. 3 are grouped together, by using the label d in the place of L and R, the encoding requires each rule to be stated separately, with its own constructor.

3.2.1 Basic properties of keys, proof labels and transitions

Before diving into the theorems related to the causality relations, our encoding requires a small library of properties of keys, proof labels and transitions: these include the *decidability of equality of keys*, the *functionality of ℓ and $\not k$*, the fact that *standard processes have no keys*, or the *loop lemma*. To provide an overview of how proofs are carried out in Beluga, we will walk through the proof of the following result: "for all proof labels θ, there exists a label α such that $\ell(\theta) = \alpha$". Its code is displayed in Fig. 7:

```
LF conn: pr_lab → pr_lab → type =
  | c_a1: conn (pr_base A K) T
  | c_a2: conn T (pr_base A K)
  | c_c1l: conn T1 T2 → conn (pr_suml T1) (pr_suml T2)
  | c_c1r: conn T1 T2 → conn (pr_sumr T1) (pr_sumr T2)
  | c_c2l: conn (pr_suml T1) (pr_sumr T2)
  | c_c2r: conn (pr_sumr T1) (pr_suml T2)
  | c_p1l: conn T1 T2 → conn (pr_parl T1) (pr_parl T2)
  | c_p1r: conn T1 T2 → conn (pr_parr T1) (pr_parr T2)
  | c_p2l: conn (pr_parl T1) (pr_parr T2)
  | c_p2r: conn (pr_parr T1) (pr_parl T2)
  | c_s1l: conn T TL → conn (pr_parl T) (pr_sync TL TR)
  | c_s1r: conn T TR → conn (pr_parr T) (pr_sync TL TR)
  | c_s2l: conn TL T → conn (pr_sync TL TR) (pr_parl T)
  | c_s2r: conn TR T → conn (pr_sync TL TR) (pr_parr T)
  | c_s3: conn T1 T1' → conn T2 T2' → conn (pr_sync T1 T2) (pr_sync T1' T2');
```

Figure 6: Encoding of the connectivity relation on proof labels.

```
LF ex_lab: pr_lab → type =
  | ex_l: lab T A → ex_lab T;
rec existence_of_lab: (g:ctx) {T:[g ⊢ pr_lab]} [g ⊢ ex_lab T] =
/ total t (existence_of_lab _ t) /
mlam T ⇒ case [_ ⊢ T] of
  | [g ⊢ pr_base _ _] ⇒ [g ⊢ ex_l lab_base]
  | [g ⊢ pr_suml T'] ⇒ let [g ⊢ ex_l L] = existence_of_lab [g ⊢ T'] in
    [g ⊢ ex_l (lab_suml L)]
  | [g ⊢ pr_sumr T'] ⇒ let [g ⊢ ex_l L] = existence_of_lab [g ⊢ T'] in
    [g ⊢ ex_l (lab_sumr L)]
  | [g ⊢ pr_parl T'] ⇒ let [g ⊢ ex_l L] = existence_of_lab [g ⊢ T'] in
    [g ⊢ ex_l (lab_parl L)]
  | [g ⊢ pr_parr T'] ⇒ let [g ⊢ ex_l L] = existence_of_lab [g ⊢ T'] in
    [g ⊢ ex_l (lab_parr L)]
  | [g ⊢ pr_sync _ _] ⇒ [g ⊢ ex_l lab_sync];
```

Figure 7: Proof of the existence of a label in a proof label.

The first two lines of code in Fig. 7 introduce a type family `ex_lab`, which captures the conclusions of the lemma to be proved: the type `ex_lab T` is inhabited whenever there exists a label `A` for which `lab T A` holds. Defining such additional type families is the standard workaround to the lack of syntactic sugar for existentials and conjunctions in Beluga.

Thanks to the Curry-Howard isomorphism, proofs by induction are encoded through recursive functions. In Beluga, these are computation-level entities introduced by the keyword `rec`. The function `existence_of_lab` takes as input a context `g` of schema `ctx` and a contextual object `T` of type `pr_lab` and returns an object of type `ex_lab T`. The second line of the proof asserts that the built function is total and is recursive on the second argument. These conditions are verified by Beluga's totality checker and guarantee that the function constitutes a valid proof.

The proof itself begins by introducing the argument `T` through the keyword `mlam`; the other argument, the context `g`, is implicit due to the use of round brackets in the function declaration. The proof proceeds by

pattern matching on the object T, which by the Curry-Howard isomorphism corresponds to case analysis on the structure of T in the informal proof. Underscores are used to omit parameters that Beluga can infer automatically. The pr_base and pr_sync cases are handled immediately, since we can already provide the required object of type ex_lab T; for the remaining four cases, the proof proceeds by recursively applying the function existence_of_lab to a subterm T' of T, and then using the result to build the desired object. Recursive calls on structurally smaller objects correspond to applications of the inductive hypothesis in the informal proof.

We conclude this subsection by addressing the *symmetry* and *irreflexivity* of the causality relations. We report the signature of the function which proves that connectivity is symmetric:

```
rec symmetric_conn: (g:ctx) [g ⊢ conn T1 T2] → [g ⊢ conn T2 T1] = ...
```

These lemmas are proved by straightforward inductions on the structure of the given predicate.

3.3 Properties of causality relations

Although the theorems in Section 2.3 are presented in a different order, here we start with the encoding of Theorem 2.6, as it is straightforward and mirrors the structure of the informal proof.

3.3.1 Encoding of Theorem 2.6

The three statements of the theorem are addressed by four recursive functions: this is because Theorem 2.6(iii) actually consists of two separate assertions, which here we prove separately. Moreover, the disjunction in the conclusions requires defining an additional type family dep_or_indep. Fig. 8 displays the proof of the final assertion: "two proof labels cannot be both dependent and independent". We also present the signatures of the other recursive functions below.

```
rec indep_impl_conn: (g:ctx) [g ⊢ indep T1 T2] → [g ⊢ conn T1 T2] = ...
rec dep_impl_conn: (g:ctx) [g ⊢ dep T1 T2] → [g ⊢ conn T1 T2] = ...
LF dep_or_indep: pr_lab → pr_lab → type =
  | or_dep: dep T1 T2 → dep_or_indep T1 T2
  | or_ind: indep T1 T2 → dep_or_indep T1 T2;
rec conn_impl_dep_or_indep: (g:ctx) [g ⊢ conn T1 T2] → [g ⊢ dep_or_indep T1 T2] = ...
```

The proof in Fig. 8 is an example of proof by contradiction: given two objects of type dep T1 T2 and indep T1 T2, the function impossible_dep_and_indep aims to derive an object of the empty type false, thereby establishing a contradiction. After introducing the arguments d and i, the proof proceeds by pattern matching on d; depending on the case, the contradiction is reached in one of three distinct ways.

In case d is built, e.g., through the constructor d_a1 (corresponding to the case A^1: $\alpha[k] \asymp \theta$ in the informal proof), it is immediately clear that an object i of type indep T1 T2 (i.e., $\alpha[k]\ \iota\ \theta$) does not exist: this contradiction is exhibited through the keyword impossible. In other subcases, such as when d is built via d_c1l (corresponding to $\mathrm{C}^1_{\mathrm{L}}$: $+_{\mathrm{L}}\theta \asymp +_{\mathrm{L}}\theta'$, given $\theta \asymp \theta'$), the contradiction is obtained by recursively invoking impossible_dep_and_indep on smaller arguments. Finally, in the d_p2l subcase ($|_{\mathrm{L}}\ \theta \asymp |_{\mathrm{R}}\ \theta'$, under the assumption $\mathscr{k}(\theta) = \mathscr{k}(\theta')$), we first examine the structure of i and find that it must have been constructed using i_p2l. This gives us an object N witnessing the inequality $\mathscr{k}(\theta) \neq \mathscr{k}(\theta')$, which clearly contradicts our assumption; however, to complete the proof in Beluga, it is first necessary to apply the auxiliary function uniqueness_of_key for some variable unification, yielding $\mathscr{k}(\theta) \neq \mathscr{k}(\theta)$, followed by the function irreflexive_neq, which states the irreflexivity of the inequality of keys.

```
rec impossible_dep_and_indep: (g:ctx) [g ⊢ dep T1 T2] → [g ⊢ indep T1 T2]
  → [g ⊢ false] =
/ total d (impossible_dep_and_indep _ _ _ d _) /
fn d,i ⇒ case d of
  | [g ⊢ d_a1] ⇒ impossible i
  | [g ⊢ d_a2] ⇒ impossible i
  | [g ⊢ d_c1l D] ⇒ let [g ⊢ i_c1l I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_c1r D] ⇒ let [g ⊢ i_c1r I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_c2l] ⇒ impossible i
  | [g ⊢ d_c2r] ⇒ impossible i
  | [g ⊢ d_p1l D] ⇒ let [g ⊢ i_p1l I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_p1r D] ⇒ let [g ⊢ i_p1r I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_p2l H1 H2] ⇒ let [g ⊢ i_p2l H1' H2' N] = i in
    let [g ⊢ refk] = uniqueness_of_key [g ⊢ H1] [g ⊢ H1'] in
    let [g ⊢ refk] = uniqueness_of_key [g ⊢ H2] [g ⊢ H2'] in irreflexive_neq [g ⊢ N]
  | [g ⊢ d_p2r H1 H2] ⇒ let [g ⊢ i_p2r H1' H2' N] = i in
    let [g ⊢ refk] = uniqueness_of_key [g ⊢ H1] [g ⊢ H1'] in
    let [g ⊢ refk] = uniqueness_of_key [g ⊢ H2] [g ⊢ H2'] in irreflexive_neq [g ⊢ N]
  | [g ⊢ d_s1l D] ⇒ let [g ⊢ i_s1l I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_s1r D] ⇒ let [g ⊢ i_s1r I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_s2l D] ⇒ let [g ⊢ i_s2l I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_s2r D] ⇒ let [g ⊢ i_s2r I] = i in impossible_dep_and_indep [g ⊢ D] [g ⊢ I]
  | [g ⊢ d_s3l D1 _] ⇒ let [g ⊢ i_s3 I1 _] = i in
    impossible_dep_and_indep [g ⊢ D1] [g ⊢ I1]
  | [g ⊢ d_s3r _ D2] ⇒ let [g ⊢ i_s3 _ I2] = i in
    impossible_dep_and_indep [g ⊢ D2] [g ⊢ I2];
```

Figure 8: Proof of the statement "two proof labels cannot be both dependent and independent".

3.3.2 Encoding of Theorem 2.1

Recall that, throughout our development, each process is assumed to be reachable. Although this hypothesis is not explicitly stated in theorems and lemmas, it is in fact essential for proving Theorem 2.1 and some of its auxiliary lemmas. For this reason, before outlining its encoding, we refine its statement, making the reachability assumption explicit:

Theorem 2.1 (Refined)

(i) If $t_1 : X_1 \overset{\theta_1}{\mapsto\!\!\!\!\rightarrow} X_1'$ *and* $t_2 : X_2 \overset{\theta_2}{\mapsto\!\!\!\!\rightarrow} X_2'$ *are connected and* X_1 *is reachable,*[2] *then* $\theta_1 \curlyvee \theta_2$.

(ii) If $\theta_1 \curlyvee \theta_2$, *then there exist* $t_1 : X_1 \overset{\theta_1}{\mapsto\!\!\!\!\rightarrow} X_1'$ *and* $t_2 : X_2 \overset{\theta_2}{\mapsto\!\!\!\!\rightarrow} X_2'$, *with* X_1 *reachable, such that* t_1 *and* t_2 *are connected.*

Since the two statements are encoded by two distinct functions, we discuss them separately. The proof of Theorem 2.1(i) is given by the following function `conn_rel_one`:

```
rec conn_rel_one: (g:ctx) {S1:[g ⊢ step X1 T1 X1']} {S2:[g ⊢ step X2 T2 X2']}
  [g ⊢ reachable X1] → [g ⊢ conn_tr S1 S2] → [g ⊢ conn T1 T2] = ...
```

[2]The reachability of the only X_1 is enough to deduce the reachability of any other process in the statement, given the existence of a path from X_1 to such processes.

The function takes as inputs two transitions `S1` and `S2`, whose typing judgments introduce the names of each involved parameter, such as the process `X1`; these are followed by the further assumptions of reachability of `X1` and connectivity of `S1` and `S2`. The function returns a derivation of the connectivity of the proof labels `T1` and `T2`.

Although our encoding may appear different – and somewhat longer – than the proof presented in [2], it is, in essence, faithful to the same underlying structure. The original proof leverages Lemma 2.3 to establish the equality of the processes O_{X_1} and O_{X_2}, the origins of the sources of the connected transitions $t_1 : X_1 \overset{\theta_1}{\mapsto\!\!\!\!\rightarrow} X_1'$ and $t_2 : X_2 \overset{\theta_2}{\mapsto\!\!\!\!\rightarrow} X_2'$. It proceeds by induction on O_{X_1}, observing that its structure determines that of the processes and transitions in the same environment (e.g., if the outermost operator of O_{X_1} is a sum, the same applies to X_1). The proof then concludes either directly or by applying the induction hypothesis to transitions involving specific subprocesses.

Below, we outline the changes and technical considerations brought by our encoding of this argument:

- The formalized proof proceeds by pattern matching on an object `D` of type `std OX1`, rather than directly on the process `OX1`. This is essentially equivalent, since the type family `std proc`, which asserts that a process is standard, is itself defined by pattern matching on the underlying process.
- It is not necessary to encode Lemma 2.3. The reachability of X_1, together with the existence of a path from X_1 to X_2, provides us a path between O_{X_1} and X_2; this path is enough to determine the structure of X_2, known the structure of O_{X_1}.
- The proof requires analyzing the structure of the given transitions `S1` and `S2`. Since combined transitions are either forward or backward, and each have their own constructors, this results in four levels of nested pattern matching. While most of the subcases can be unified in the informal proof, Beluga requires them to be treated separately: this is the primary reason for the proof's length. To improve efficiency, certain assertions have been moved earlier in the proof tree compared to their position in the informal version.
- The informal proof takes for granted structural properties such as: "given a path whose source is a sum process, the target is also a sum process", or "given a path between two sum processes, there exists a path between their left addends". In the encoding, these results must be explicitly stated and proved, resulting in 16 additional lemmas. Some of these require classical techniques such as mutual recursion or strengthening of contextual judgments, which are described in [8]. We report the signatures of two of these functions:

```
% Type family encoding sum processes
LF is_sum: proc → type =
  | sm: is_sum (sum X Y);
% A path starting from a sum process ends in a sum process
rec step*_from_sum: (g:ctx) [g ⊢ step* (sum X Y) Z] → [g ⊢ is_sum Z] = ...
% Given a path between sum processes, there is a path between their left addends
rec step*_betw_sums_left: (g:ctx) [g ⊢ step* (sum X1 X2) (sum Y1 Y2)]
  → [g ⊢ step* X1 Y1] = ...
```

The formalization of Theorem 2.1(ii) requires encoding Lemma 2.5, which states that every proof label θ is realised by some process $r(\theta)$ – that is, there exist processes X_1, X_2 and $r(\theta)$ such that $r(\theta) \mapsto\!\!\!\!\rightarrow^* X_1 \overset{\theta}{\mapsto\!\!\!\!\rightarrow} X_2$. The original proof in [2], however, goes further: it builds a process $r(\theta)$ which is standard and directly performs a single forward transition $r(\theta) \overset{\theta}{\mapsto} X_2$ (in other words, $r(\theta)$ and X_1 coincide). Our encoding reflects this stronger formulation by specializing the original Definition 2.4 with the following type family `realised`:

```
LF realised: pr_lab → type =
  | rl: std X → fstep X T X' → realised T;
```

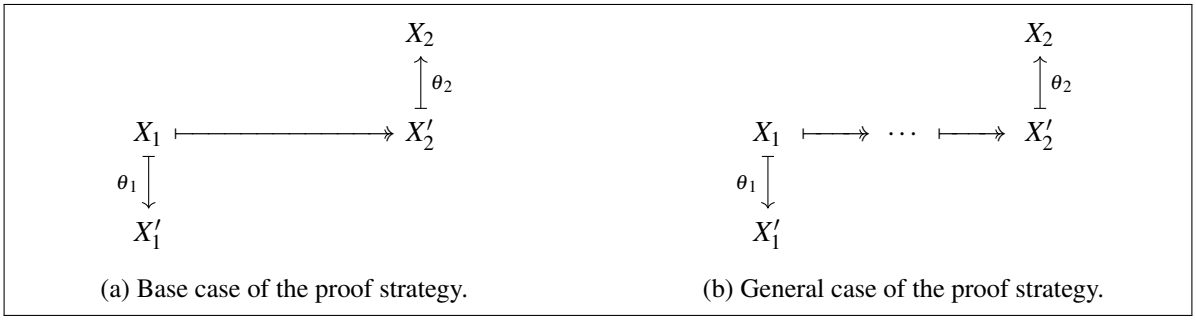
For any proof label `T`, `realised T` is non empty iff `std X` and `fstep X T X'` hold for some `X` and `X'`. The following recursive function `pr_lab_is_realised` encodes the proof of Lemma 2.5:

```
rec pr_lab_is_realised: (g:ctx) [g ⊢ valid T] → [g ⊢ realised T] = ...
```

The proof is a straightforward induction on the structure of the assumption `valid T`.

Other than relying on Lemma 2.5, the proof of Theorem 2.1(ii) in [2] assumes auxiliary results such as the following: "if O_X realises X and O_Y realises Y, then $O_X \mid O_Y$ realises $X \mid Y$". While this result holds in the particular context of Theorem 2.1(ii), where $X \mid Y$ is known to be reachable and is able to perform a synchronization, it does not hold in general. For instance, consider $X_1 = a[k]$ and $X_2 = b[k]$: the parallel composition $a[k] \mid b[k]$ is not reachable from $a \mid b$. Moreover, even when such conditions are met, building a constructive proof is far from straightforward. These issues led us to revisit the entire argument and develop the following proof strategy for Theorem 2.1(ii):

1. First, we consider the case where neither $\ell(\theta_1)$ nor $\ell(\theta_2)$ is τ and prove that the diagram in Fig. 9a holds. The hypothesis excludes the cases in which θ_1 and θ_2 label synchronizations, thus ruling out the scenarios in which the aforementioned auxiliary lemma occurs. The proved result goes beyond establishing the connectivity of two combined transitions labelled by θ_1 and θ_2: both transitions are forward, and the processes X_1 and X_2' are either identical or connected by a single combined transition. Additionally, we show that at least one among X_1 and X_2' is standard.
2. We then move to the general case, proving that for any connected pair of proof labels θ_1 and θ_2 the diagram in Fig. 9b holds. Analogously to the previous point, the transitions labelled by θ_1 and θ_2 are forward, meaning that our statement is slightly more specific than the original formulation of Theorem 2.1(ii). This refinement helps eliminating non-existent subcases that would arise in the nested pattern matching of combined transitions.



(a) Base case of the proof strategy.

(b) General case of the proof strategy.

Figure 9: Proof strategy for Theorem 2.1(ii).

In the general case, when θ_1 and θ_2 label synchronizations (e.g., when $\theta_1 = \langle |_{\mathrm{L}}\theta_L^1, |_{\mathrm{R}}\theta_R^1 \rangle$), the labels of their subterms (e.g., θ_L^1 and θ_R^1) are not τ: this detail allows us to apply the base case of the proof strategy, which provides richer information than the inductive hypothesis of the general Theorem 2.1(ii). That additional information is essential: it enables us to build the desired path between X_1 and X_2', actually with at most two transition steps.

The encoding of Theorem 2.1(ii) follows the plan outlined. The base case makes use of the following elements: a type family `lab_not_tau`, characterizing proof keyed labels whose label is not τ;

a type family max_one_step, encoding the conclusions of the statement; and the recursive function conn_rel_two_base, which proves it.

```
LF lab_not_tau: pr_lab → type =
  | nt_inp: lab T (inp _) → lab_not_tau T
  | nt_out: lab T (out _) → lab_not_tau T;
LF max_one_step: pr_lab → pr_lab → type =
  | c_id: std X1 → fstep X1 T1 X1' → fstep X1 T2 X2 → max_one_step T1 T2
  | c_fw: std X1 → fstep X1 T1 X1' → fstep X1 T3 X2' → fstep X2' T2 X2
      → lab T1 L1 → lab T3 L1 → max_one_step T1 T2
  | c_bw: std X2' → fstep X1 T1 X1' → bstep X1 T3 X2' → fstep X2' T2 X2
      → lab T2 L2 → lab T3 L2 → max_one_step T1 T2;
rec conn_rel_two_base: (g:ctx) [g ⊢ valid T1] → [g ⊢ valid T2] → [g ⊢ conn T1 T2] →
  [g ⊢ lab_not_tau T1] → [g ⊢ lab_not_tau T2] → [g ⊢ max_one_step T1 T2] = ...
```

The proof of this result is given by a long induction on the structure of the given connectivity relation. The predicates (lab Ti Lj), for i,j in $\{1,2,3\}$, which occur in the type family max_one_step, are a technical detail which helps completing few subcases of the proof.

Next, the general case of the proof is addressed by the recursive function conn_rel_two_fstep below, which relies on a dedicated type family as well:

```
LF ex_conn_fstep: pr_lab → pr_lab → type =
  | ex_cf: fstep X1 T1 X1' → fstep X2' T2 X2 → step* X1 X2'
      → reachable X1 → ex_conn_fstep T1 T2;
rec conn_rel_two_fstep: (g:ctx) [g ⊢ valid T1] → [g ⊢ valid T2] → [g ⊢ conn T1 T2]
  → [g ⊢ ex_conn_fstep T1 T2] = ...
```

The proof is given by a long induction on the structure of the given connectivity relation. It requires encoding auxiliary lemmas such as the following: "given a path between two processes X and X', there is a path between $X+\mathbf{0}$ and $X'+\mathbf{0}$", or: "given a forward transition $X \overset{\theta}{\mapsto} X'$ where X is standard and $\not{k}(\theta) = k$, then any key $m \neq k$ does not occur in X'".

Finally, Theorem 2.1(ii) is encoded by the following function conn_rel_two. It calls the function conn_rel_two_fstep, applies the loop lemma to reverse one of the two forward transitions, and has all the ingredients to conclude:

```
LF ex_conn_tr: pr_lab → pr_lab → type =
  | ex_c: {S1: step X T1 X'} {S2: step Y T2 Y'} reachable X → conn_tr S1 S2
      → ex_conn_tr T1 T2;
rec conn_rel_two: (g:ctx) [g ⊢ valid T1] → [g ⊢ valid T2] → [g ⊢ conn T1 T2]
  → [g ⊢ ex_conn_tr T1 T2] =
/ total c (conn_rel_two _ _ _ _ _ c) /
fn v1,v2,c ⇒
let [g ⊢ ex_cf F1 F2 S* (rch D S0*)] = conn_rel_two_fstep v1 v2 c in
let [g ⊢ B2] = loop_lemma_one [g ⊢ F2] in
[g ⊢ ex_c (fw F1) (bw B2) (rch D S0*) (ct (fw F1) (bw B2) S*)];
```

4 Conclusions and Future Work

We begin with a brief technical overview of the encoding. The complete formalization consists of less than 2000 lines of code and includes a total of 49 theorems and lemmas. Among them, 13 are direct

translations of results stated in Section 2, while the remaining 36 are technical and auxiliary lemmas introduced to support the encoding.

Beluga has proved to be a reliable and expressive proof assistant, well-suited to represent the definitions and properties of CCSK^P. Its use of higher-order abstract syntax (HOAS) offers a convenient approach to handling restrictions – even though CCSK^P does not feature a particularly complex binding structure, unlike, for instance, the π-calculus. Furthermore, Beluga's explicit proof style provides a transparency that is often lost in proof assistants that rely heavily on automation.

However, the lack of automation also comes with drawbacks, mainly the increased length of proof terms. This also follows from the lack of syntactic sugar for existentials, conjunctions and disjunctions, which leads to defining additional type families or splitting theorem statements. Additionally, Beluga provides no built-in mechanism to simplify repeated proof patterns, requiring each similar subcase to be handled individually.

Whether the overall outcome is favorable depends largely on the specific system one aims to formalize. For languages with rich binding structures, the benefits of HOAS alone may outweigh the trade-offs. In our case, however, this advantage is less significant, and we believe that other proof assistants (such as Rocq [5]) might be a better fit for formalizing the system at hand.

To the best of our knowledge, this work provides the first formalization of a reversible concurrent calculus in a proof assistant. We have formally verified the correctness of the notions and results presented in Section 2. We gained a deeper understanding of the system itself, leading to refinements in both definitions and proofs; in particular, we provided an alternative way to represent proof labels compared to the informal definition.

This work lays the foundations for future reversible concurrent calculi formalizations. The encoding can be adapted to cover the subsystems of CCSK^P, i.e., CCS and CCSK, and can be mapped to existing CCS formalizations. Moreover, it could be extended to include additional portions of [3]. Additionally, it could be translated into other proof assistants, such as Rocq, which are potentially better suited for representing this reversible process calculus. Finally, it can serve as a reference point for future formalizations of other reversible concurrent calculi, such as RCCS.

Acknowledgments

We warmly thank the anonymous reviewers, as well as Clément Aubert and Deivid Vale, for their insightful comments and suggestions, which greatly contributed to improving the paper. This work is supported by the National Science Foundation under Grant No. 2242786 (SHF:Small:Concurrency In Reversible Computations).

References

[1] Clément Aubert & Peter Browning (2023): *Implementation of a Reversible Distributed Calculus*. In Martin Kutrib & Uwe Meyer, editors: *Reversible Computation - 15th International Conference, RC 2023, Giessen, Germany, July 18-19, 2023, Proceedings*, *Lecture Notes in Computer Science* 13960, Springer, pp. 210–217, doi:10.1007/978-3-031-38100-3_13.

[2] Clément Aubert, Iain Phillips & Irek Ulidowski (2024): *Dependence and Independence for Reversible Process Calculi*. *CoRR* abs/2410.14699, doi:10.48550/ARXIV.2410.14699.

[3] Clément Aubert, Iain Phillips & Irek Ulidowski (2025): *Independence and Causality in the Reversible Concurrent Setting*. In Robert Glück & Robin Kaarsgaard, editors: *Reversible Computation - 17th International*

Conference, RC 2025, Odense, Denmark, July 3-4, 2025, Proceedings, *Lecture Notes in Computer Science* 15716, Springer, pp. 9–26, doi:10.1007/978-3-031-97063-4_2.

[4] Clément Aubert (2022): *Concurrencies in Reversible Concurrent Calculi*. In Claudio Antares Mezzina & Krzysztof Podlaski, editors: *Reversible Computation - 14th International Conference, RC 2022, Urbino, Italy, July 5-6, 2022, Proceedings*, *LNCS* 13354, Springer, pp. 146–163, doi:10.1007/978-3-031-09005-9_10.

[5] 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.

[6] Marco Carbone, David Castro-Perez, Francisco Ferreira, Lorenzo Gheri, Frederik Krogsdal Jacobsen, Alberto Momigliano, Luca Padovani, Alceste Scalas, Dawit Tirore, Martin Vassor, Nobuko Yoshida & Daniel Zackon (2024): *The Concurrent Calculi Formalisation Benchmark*. In Ilaria Castellani & Francesco Tiezzi, editors: *Coordination Models and Languages*, Springer Nature Switzerland, Cham, pp. 149–158, doi:10.1007/978-3-031-62697-5_9.

[7] Gabriele Cecilia (2025): *A Formalization of the Reversible Concurrent Calculus CCSK^P in Beluga (artifact)*, doi:10.5281/zenodo.16179366.

[8] Gabriele Cecilia & Alberto Momigliano (2024): *A Beluga Formalization of the Harmony Lemma in the π-Calculus*. In Florian Rabe & Claudio Sacerdoti Coen, editors: Proceedings Workshop on *Logical Frameworks and Meta-Languages: Theory and Practice*, Tallinn, Estonia, 8th July 2024, *Electronic Proceedings in Theoretical Computer Science* 404, Open Publishing Association, pp. 1–17, doi:10.4204/EPTCS.404.1.

[9] James Cheney, Michael Norrish & René Vestergaard (2012): *Formalizing Adequacy: A Case Study for Higher-order Abstract Syntax*. *J. Autom. Reason.* 49(2), pp. 209–239, doi:10.1007/S10817-011-9221-6.

[10] Gavin Cox (2009): *SimCCSK: simulation of the reversible process calculi CCSK*. Master's thesis, University of Leicester. Available at `https://figshare.le.ac.uk/articles/thesis/SimCCSK_simulation_of_the_reversible_process_calculi_CCSK/10091681`.

[11] Vincent Danos & Jean Krivine (2004): *Reversible Communicating Systems*. In Philippa Gardner & Nobuko Yoshida, editors: *CONCUR 2004*, *LNCS* 3170, Springer, pp. 292–307, doi:10.1007/978-3-540-28644-8_19.

[12] Pierpaolo Degano & Corrado Priami (2001): *Enhanced operational semantics*. *ACM Comput. Surv.* 33(2), pp. 135–176, doi:10.1145/384192.384194.

[13] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics*. *J. ACM* 40(1), pp. 143–184, doi:10.1145/138027.138060.

[14] Furio Honsell, Marino Miculan & Ivan Scagnetto (2001): *π-Calculus in (Co)Inductive Type Theory*. *Theor. Comput. Sci.* 253(2), pp. 239–285, doi:10.1016/S0304-3975(00)00095-5.

[15] T. F. Melham (1994): *A Mechanized Theory of the π-Calculus in HOL*. *Nordic J. of Computing* 1(1), p. 50–76, doi:10.48456/tr-244.

[16] Robin Milner (1980): *A Calculus of Communicating Systems*. LNCS, Springer-Verlag, doi:10.1007/3-540-10235-3.

[17] Robin Milner, Joachim Parrow & David Walker (1992): *A Calculus of Mobile Processes, I*. *Inf. Comput.* 100(1), pp. 1–40, doi:10.1016/0890-5401(92)90008-4.

[18] Monica Nesi (1992): *A formalization of the process algebra CCS in high order logic*. Technical Report UCAM-CL-TR-278, University of Cambridge, Computer Laboratory, doi:10.48456/tr-278.

[19] Frank Pfenning & Conal Elliott (1988): *Higher-Order Abstract Syntax*. In: *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, ACM Press, pp. 199–208, doi:10.1145/53990.54010.

[20] Iain Phillips & Irek Ulidowski (2007): *Reversing algebraic process calculi*. *J. Log. Algebr. Program.* 73(1-2), pp. 70–96, doi:10.1016/j.jlap.2006.11.002.

[21] Brigitte Pientka & Jana Dunfield (2010): *Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)*. In Jürgen Giesl & Reiner Hähnle, editors: *Automated Reasoning*,

5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings, *Lecture Notes in Computer Science* 6173, Springer, pp. 15–21, doi:10.1007/978-3-642-14203-1_2.

[22] Irek Ulidowski, Iain Phillips & Shoji Yuen (2014): *Concurrency and Reversibility*. In Shigeru Yamashita & Shin-ichi Minato, editors: *Reversible Computation - 6th International Conference, RC 2014, Kyoto, Japan, July 10-11, 2014. Proceedings*, *LNCS* 8507, Springer, pp. 1–14, doi:10.1007/978-3-319-08494-7_1.